

Programming with VisualAge for Java

The VisualAge Series



Bitterer, Brassard, Nadal, and Wong

VisualAge and Transaction Processing in a Client/Server Environment

Bitterer, Hamada, Oosthuizen, Porciello, and Rambek

AS/400 Application Development with VisualAge for Smalltalk

Bitterer and Carrel-Billiard

World Wide Web Programming: VisualAge for C++ and Smalltalk

Carrel-Billiard and Akerley

Programming with VisualAge for Java

Carrel-Billiard, Jakab, Mauny, and Vetter

Object-Oriented Application Development with VisualAge for C++ for OS/2

Carrel-Billiard, Friess, and Mauny

Programming with VisualAge for C++ for Windows

Fang, Chu, and Weyerhäuser

VisualAge for Smalltalk and SOMObjects: Developing Distributed Object Applications

Fang, Guyet, Haven, Vilmi, and Eckmann

VisualAge for Smalltalk Distributed: Developing Distributed Object Applications

Nilsson and Jakab

VisualAge for C++: Visual Programmer's Handbook

Programming with **VisualAge for Java**

Marc Carrel-Billiard
John Akerley



INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION
SAN JOSE, CALIFORNIA 95120

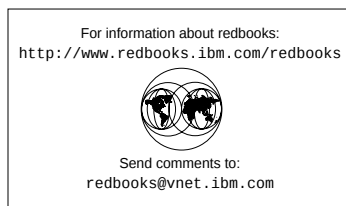
PRENTICE HALL PTR
UPPER SADDLE RIVER, NEW JERSEY 07458

© Copyright International Business Machines Corporation 1998. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

This edition applies to Version 1.0 of the VisualAge for Java product set, and to all subsequent releases and modifications until otherwise indicated in new editions.

Comments about ITSO Technical Bulletins may be addressed to:
IBM Corporation ITSO, 471/80-E2, 650 Harry Road, San Jose, California 95120-6099



Published by Prentice Hall PTR
Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, NJ 07458

Acquisitions Editor: Michael E. Meehan

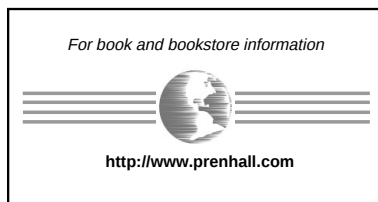
Manufacturing Manager: Alexis R. Heydt

Cover Design: Andreas Bitterer

Copy Editor: Maggie Cutler

Production Supervision: Craig Little

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:
Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458
Phone: 800-382-3419; FAX: 201-236-7141; E-mail (Internet): corpsales@prenhall.com



Printed in the United States of America

1 0 9 8 7 6 5 4 3 2 1

ISBN 0-13-911371-1

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To my family: Dominique, Thomas, and Fanny.

Marc

To Jasna for writing and editing insight and, more importantly, support.

John

Foreword

When we set about to develop a VisualAge programming environment for Java, we first needed to understand the kinds of applications that businesses wanted to build, applications that could leverage the capability that Java brought to the Web. We discovered that businesses wanted to do more than add graphical appeal to their web sites, they wanted to extend their enterprise reach to the web. More than Web advertising and simple CGI scripts, businesses wanted to create real applications that leveraged their existing environments and data. VisualAge for Java accepts this business goal and is making it a reality today for our customers, large and small.

The development environment itself is object-based and tied closely to the dynamic nature of Java to support incremental compilation and debug, plus automatic version control down to the method level. In this book you will learn all about VisualAge for Java, get the Java basics, work through real programming examples that explain the Java programming language, JavaBeans, user interface programming and explore visual programming with the VisualAge visual composition editor. Developing applets and JavaBeans through visual programming and code generation make this a powerful rapid application development environment for Java.

Marc and John are recognized by the development team as contributors to VisualAge. Their work on this book will enhance your Java programming skills and your experience with VisualAge for Java.

The CD-ROM that accompanies the book contains VisualAge for Java Entry version. This version is a full function edition of the product that is only limited by the total number of classes you can create. Along with this book I am sure VisualAge for Java Entry will launch you into the exciting world of Java and the Internet. Install it on your own system and give it a try.

Welcome to the world of VisualAge!

Paul Buck
Solution Manager - IBM VisualAge for Java
IBM Corporation

Preface

Welcome to the world of visual programming with Java! With VisualAge for Java, you are ready to take the plunge into a radically new way of programming. If you have just purchased your IBM VisualAge for Java and are dying to build your first application or applet, you are reading the right book. Indeed, learning VisualAge for Java by example is this book is all about. With VisualAge for Java, application construction has never been easier. Even the most complex applications can be constructed from the large set of predefined Java beans. This book, the first in the VisualAge for Java series, shows you how to get started with IBM VisualAge for Java.

What Is Java?

Less than a year ago Java “spilled” into the world of network computing (Coffee-related metaphor is a big part of the Java scene!). An object-oriented programming language, Java was designed from the ground up as a secure, portable, and efficient language for programming appliances and personal digital assistants (PDAs). Java did not quite take off in this market place however, and for this reason was repurposed for the Web by Sun Microsystems. If you are reading this book this means that you care about Java and that is good because Java may change the Web profoundly as well as the programming of client/server applications.

Who Should Read This Book?

This book is best used as a starting point for learning Java in the VisualAge for Java environment. If you do not know object-oriented programming, this book will help you get started with a powerful way of developing your applications. For readers who already have some knowledge of Java, this book has been organized such that you can easily find the parts that describe how to use VisualAge for Java.

Let's Get Practical

This book is not a complete Java reference book. It does not have complete descriptions and usage examples of all Java packages. Instead, this book concentrates on how to use VisualAge for Java to build Java classes and beans. If you are looking for a practical

guide on how to program Java with VisualAge for Java, look no further, you are holding the book you need. Through 15 chapters, this book takes you on a Java journey. During the trip, you design and program an ATM applet from the ground up, modeling and implementing its logic using object-oriented principles, sketching its view or user interface on the chalkboard, and building it visually with the Visual Composition Editor. The trip ends with you assembling the application's model and view to complete your final applet.

How This Book Is Organized

This book consists of three parts. In the first part (Chapters 1 through 9) we introduce the Java language and the object-oriented concepts and principles you need to know to program in Java. We also introduce the IBM VisualAge for Java integrated development environment (IDE), which you learn to use in each chapter. In the second part (Chapters 10 through 12), we introduce the JavaBeans specification, which describes the Java component software that IBM VisualAge for Java uses to support the IBM award-winning construction from parts paradigm. You learn about visual programming in Java through the Visual Composition Editor of VisualAge for Java. In the last part (Chapters 13 through 15), we show you how to improve and publish your applet. We conclude with a short overview of the Enterprise edition of IBM VisualAge for Java.

In this book each chapter is divided in two parts, the first concentrating on Java as a language, and the second concentrating on how to implement what you have learned, using VisualAge for Java. Here is an overview of each chapter:

❑ **Chapter 1, “Introduction to the Environment”**

The first chapter is for those who cannot wait! We quickly introduce VisualAge for Java, and in just a few steps you are able to build your first applets and applications.

❑ **Chapter 2, “Java Basics”**

If you are new to the Java language, read this chapter for a crash course on the language and its syntax. In the second part of this chapter you explore the VisualAge for Java IDE and learn how to test your code in the Scrapbook.

❑ **Chapter 3, “Objects and the Java Language”**

This chapter introduces you to objects and shows you how you can use objects in Java. You learn your first object-oriented concepts, such as objects, classes, interfaces, messages, and encapsulation. To illustrate the new concepts, you build your

first class, a bank account class, using the VisualAge for Java SmartGuides. You refine the bank account class throughout the book.

❑ **Chapter 4, “Organizing Your Code”**

Well-organized code can save you time and prevent headaches! This chapter introduces you to Java packages and VisualAge projects which help you organize your code. You also learn to use SmartGuides to import and export code to and from the VisualAge for Java IDE.

❑ **Chapter 5, “Lifecycle of Java Objects”**

To exist, objects must be created first. In this chapter you learn how to create objects and how to get rid of them when you no longer need them. Using the VisualAge for Java SmartGuides and the Scrapbook, you can put into practice what you have learned.

❑ **Chapter 6, “Reuse in Java”**

If you want to grasp the key concepts of object-oriented programming, this chapter is your open Sesame! Using the ATM example, you learn how to achieve code reuse, using inheritance and aggregation relationships between objects. You also learn how polymorphism can help you write better code. In the second part of this chapter, you use VisualAge for Java to create more classes and expand the ATM model.

❑ **Chapter 7, “Error Handling and Debugging”**

Run-time errors can occur unexpectedly, and handling them can be a very tedious task when developing an application. Fortunately Java has an innovative approach that enables you to cleanly incorporate error handling from the start in your code. This chapter describes that approach and introduces you to the powerful VisualAge for Java debugger, which helps you track even the most tricky bugs. As always, you practice the new concepts by providing your ATM model with a new exception class and the mechanism for handling it.

❑ **Chapter 8, “File I/O and Persistence”**

Like most programming languages, Java provides your programs with facilities to support input and output through the console or using files. Input and output streams are introduced in this chapter along with a powerful mechanism, serialization, that enables you to save objects state and restore it later on. Using serialization, you transform your ATM model to make it persistent.

❑ **Chapter 9, “Managing Your Code”**

Modifying your code is one thing, retrieving the modification history is another. With VisualAge for Java every modification you make in your code is kept in a underlying database. Thus you can backtrack your modifications and restore old versions of your code. This chapter describes the versioning mechanism provided by VisualAge for Java and how you can take advantage of it in the implementation of your ATM applet.

❑ **Chapter 10, “Building User Interfaces with Java”**

Once your ATM model is built, you must provide it with a view that represents the user interface of your applet. This chapter shows you how to use the Abstract Windowing Toolkit (AWT) to create user interfaces in Java. Several small applications are built to illustrate the concepts learned.

❑ **Chapter 11, “Visual Programming and JavaBeans”**

With the new release of the Java Developer’s Toolkit (JDK 1.1), a new software component specification, called *JavaBeans*, is introduced. This chapter explains Java software components, or beans, and shows how you can build your own beans using the VisualAge for Java Visual Composition Editor. Visual programming is also introduced. To illustrate the concepts and techniques described in this chapter, you build a calculator that you reuse later in your ATM applet.

❑ **Chapter 12, “Visual Programming in Action”**

After you have grasped the power of JavaBeans and the visual construction from beans provided by the Visual Composition Editor, we show you the techniques that you must master to build a user interface visually. You practice by building the view of your ATM applet that you assemble with the ATM model to compose the final applet.

❑ **Chapter 13, “Improving Your ATM Applet”**

In this chapter you learn about multithreading, run-time type information handling with the Visual Composition Editor, and internationalization of your program. These advanced features are illustrated by improving your applet.

❑ **Chapter 14, “Publishing Your Applet”**

Once your applet has been fully tested, you may want to publish it on the Internet. This chapter shows you how to do that—from writing an HTML page that embeds your applet, to setting up your Web server.

❑ Chapter 15, “VisualAge for Java Enterprise Edition”

This chapter provides a sneak preview of the Enterprise edition of VisualAge for Java, which enables you to automatically generate beans that connect to your legacy data and code. ATM applet improvements are discussed in regard to these new possibilities.

CD-ROM Included



The CD-ROM that accompanies this book contains the IBM VisualAge for Java Entry Edition for the Windows 95/NT and OS/2 platforms. This edition is a scaled down version of the Professional edition limited to 100 classes. The CD-ROM also contains the code samples described throughout the book and the ATM applet. Refer to the README file in the root directory of the CD-ROM for the latest information about installing VisualAge for Java and the sample code on your machine.

Although the illustrations in this book have been captured from a Windows platform, the VisualAge for Java product has the same look and feel on OS/2, and the samples described are compatible with both the Windows and OS/2 versions of IBM VisualAge for Java.

Special Conventions in This Book

Several boxes with predefined icons are used throughout the book to point out important information.



Tips

The helpful tips scattered throughout the book are designed to make your life easier.



Warning

Be sure to read the warning boxes. They are designed to draw your attention to areas where you can get into trouble.



Read This

The Read This boxes point out information that you should not miss while reading the chapters and developing your sample application.



Information

The technical information in these boxes complements the information we provide throughout the book.

ITSO on the Internet



Internet users can find information about redbooks on the ITSO World Wide Web (WWW) home page. To access the ITSO Web pages, point your Web browser (such as WebExplorer™ from the OS/2 3.0 Warp BonusPak) to the following:

`http://www.redbooks.ibm.com/redbooks`

IBM internal users can download redbooks or read redbooks online. Point your web browser to the internal IBM Redbooks home page:

`http://w3.itso.ibm.com/redbooks`

If you do not have World Wide Web access, you can obtain the list of all current redbooks through the Internet by anonymous FTP to:

```
ftp.almaden.ibm.com
cd /redbooks
get itsopub.txt
```

The FTP server, *ftp.almaden.ibm.com*, also stores the sample from the accompanying CD-ROM. To retrieve the sample files, issue the following commands from the */redbooks* directory:

```
cd GG242232
binary
get GG242232.ZIP
ascii
get READ.ME
```

All users of ITSO publications are encouraged to provide feedback to improve quality over time. Send questions about and feedback on redbooks to:

`redbook@vnet.ibm.com` or
REDBOOK at WTSCPOK or
USIB5FWN at IBMMAIL

To receive regular updates on new redbooks and general IBM announcements, you can subscribe to the IBM Announcement Listserver. It automatically supplies an Internet e-mail user with timely new announcement information (titles and optionally the letter or abstract) from selected categories. To get started, send an e-mail to:

`announce@webster.ibm.link.ibm.com`

The keyword “SUBSCRIBE” must be the only word in the body of the e-mail; leave the subject line blank. You will receive a category form and listserver details. To immediately start your subscription of, for example, AS/400 announcements, put the words “SELECT HW120” in the body of the note. On the afternoon of an announcement day, you will receive e-mail with the announcement, along with a list of newly available redbook titles. To obtain a full abstract of a particular redbook, use “GET SG242232” in the note.

VisualAge for Java Support

VisualAge for Java Service and Support is staffed by developers who handle everything from how-tos to complex technical problems. The resolution may take the form of education, a work-around, or a fix to the product (Corrective Service Diskette, CSDs).

There are several ways to contact the VisualAge for Java Service and Support department electronically:

- ❑ CompuServe™ forums: GO VAJAVA
- ❑ Internet
 - anonymous logon to:
 - site:* `ftp.software.ibm.com`
 - directory:* `ps/products/visualagejava/fixes/`
 - sample URL:
`ftp://ftp.software.ibm.com/ps/products/visualage-java/fixes/`
 - Web site:
`http://www.software.ibm.com/ad/vajava`
- ❑ Talklink
 - 1-800-992-4777 for information (USA and Canada)
 - VAJAVA CFORUM
- ❑ Developer's Connection (DEVCON) CD
 - Ordering information: 1-800-6DE-VCON (USA and Canada)
- ❑ IBM PC Co.: BBS 1-919-517-0001 (8,N,1) or 1-800-772-2227 for information

About the Authors

Marc Carrel-Billiard, is a consultant at the IBM International Technical Support Organization's Almaden Research Center in San Jose, California. He holds a Master's of Computing Science from Ecole Supérieure des Telecommunications of Paris, France. He has over 15 years of experience as a software architect and project manager, and has taught skill-transfer workshops worldwide on client/server, object-oriented and Java application development. Marc is co-author of *Programming with VisualAge for C++ for Windows* and *World Wide Web Programming with VisualAge for C++ and Smalltalk* (Prentice Hall PTR). You can reach Marc by e-mail at carrel@vnet.ibm.com.

After completing a Master's of Computing Science in 1989 John Akerley went on to design UNIX courseware and develop a non-linear video editing system. He joined IBM in 1992 and wrote, developed and taught in the AIX CASE area. In 1994 he began building web applications and since February of 1996 he has been working with Java. As well as Java consulting and teaching VisualAge for Java courses and workshops he currently works on the VisualAge for Java certification program. You can reach John by e-mail at akerley@ca.ibm.com.

Acknowledgments



This book would not have been possible without the help of the following people, who contributed to the contents of this book:

- ❑ Dominique Faidherbe from IBM Belgium designed a first version of the ATM applet and wrote about the Visual Composition Editor.
- ❑ Jouko Ruuskanen from IBM Finland wrote about VisualAge for Java IDE and code versioning.
- ❑ Werner Schnedl from Coopers and Lybrand Switzerland wrote about object-oriented principles and Java exception handling.
- ❑ Achim Hasenmüller from IBM France provided valuable information about the JDK1.1 internationalization framework.

Many thanks to Carmine Di Grande, ITSO San Jose Center AD Manager, and Petter Sommerfelt, ITSO San Jose Center DM/ST Manager for getting this project started. Special thanks to everyone at the ITSO San Jose Center, in particular Elsa Barron and Mary Comianos, for their continuous support. Thanks to the talented Andi Bitterer, who designed the cover of this book and created a masterpiece! We are extremely grateful to Maggie Cutler for her meticulous and unfailing editorial review. Thanks to Ueli

Wahli from ITSO San Jose, Peter Jakab, Mark Ryan and Barry Searle from IBM Canada Toronto Lab for testing the code and providing a thorough technical review of the book. Thanks also to Mike Meehan at Prentice Hall for making this book happen.

M.C.B.

J.A.

Acknowledgments

Contents

Foreword	vi
Preface	vii
What Is Java?	vii
Who Should Read This Book?.....	vii
Let's Get Practical.....	vii
How This Book Is Organized	viii
CD-ROM Included.....	xi
Special Conventions in This Book	xi
ITSO on the Internet.....	xii
VisualAge for Java Support	xiii
About the Authors.....	xiv
Acknowledgments	xiv
Figures	xxv
Tables	xxix

Part 1. VisualAge for Java and Java Basics

Chapter 1. Introduction to the Environment	3
The VisualAge for Java Development Environment.....	3
Building Your First Applet	5
Let's Get Started!.....	5
The Workbench Explained.....	10
Creating An Animated Applet.....	12
Building Your First Application	13
VisualAge for Java Symbols Explained	16
Chapter 2. Java Basics	19
How Java Works	19
The Java Language.....	20
Statements	21
Variables, Keywords, and Operators.....	21
Comments.....	29
Predefined Classes: String and Array	29
Control Flow.....	31
The VisualAge for Java Scrapbook.....	35
Using the Scrapbook	35
Correcting Errors in the Scrapbook	38

Chapter 3. Objects and the Java Language	41
Object-Oriented Programming Concepts	42
Objects	42
Messages	45
Classes	46
What Makes Java Object-Oriented?	46
The Automated Teller Machine (ATM)	47
System Requirements	48
Analysis	48
Design	50
Implementation	50
The BankAccount Class	51
Creating Java Classes	51
Fields	52
Methods	53
Wrapper Classes	56
The BankAccount Objects	56
Static Fields and Static Methods	57
Using Objects within Your Programs	58
Object References	58
Message Sending	59
Object Assignment	59
Implementing the BankAccount Class	60
Creating a New Class	60
Creating the Balance Field	62
Creating the AccountId Field	63
Creating Getter and Setter Methods for the Balance Field	63
Creating the Getter and Setter Methods for the AccountId Field	65
Correcting Mistakes	65
Creating Debit and Credit Methods	66
Testing the Class in the Scrapbook	69
Chapter 4. Organizing Your Code	71
What Is a Package?	71
More about Access Modifiers	72
Packages in File-Based Java	73
CLASSPATH Environment Variable	73
Creating Packages	73
Using Packages	74
Packages in VisualAge for Java	75
Creating Packages	76
Using Packages in Your VisualAge for Java Code	77
Importing and Exporting Java Code	78
Importing Java Code to VisualAge for Java	78
Using the Import SmartGuide	80
Exporting Java Code from VisualAge for Java	81
Projects in VisualAge for Java	82
Java Class Libraries	83

Chapter 5. Lifecycle of Java Objects	85
Instantiating Objects	85
Initializing Fields	86
Constructors	87
Overloading a Constructor	89
The this Keyword	90
Copy Constructor and References	90
Passing Arguments in Methods	93
Access Modifiers and Constructors	94
When the Work Is Done: Finalizers	94
Creating Constructors and Finalizers with VisualAge for Java	95
Copying Packages	96
Adding the initialAmount Field	96
Creating Constructors with VisualAge for Java	97
Defining Finalizers with VisualAge for Java	100
 Chapter 6. Reuse in Java	 103
Reuse in Object-Oriented Languages	103
Is-A and Has-A Relationships	104
Inheritance	104
Do You Speak Object-Oriented?	104
Creating Subclasses with Java	106
Incremental Development	106
Abstract Classes	107
Abstract Methods	108
Polymorphism	109
Interfaces	112
Interfaces and Abstract Classes	114
Aggregation	115
Banking Transactions	115
Delegation	117
Association	118
Inheritance and Aggregation	118
Inner Classes	118
Class Implementation Revisited	119
Modifiers Revisited	119
Constructors Revisited	120
Reuse with VisualAge for Java	121
Implementing Inheritance	121
Implementing Abstract Classes	128
Implementing an Interface	130
Implementing Aggregation	133
 Chapter 7. Error Handling and Debugging	 139
Overview of Exception Handling in Java	139
Why a New Error Handling Mechanism?	140
What Are Exceptions?	141

Exception Handling Keywords and Concepts	141
Java Standard Exceptions	146
The Throwable Hierarchy	146
Using Predefined Exceptions	148
Creating and Throwing Your Own Exceptions	149
Defining a New Exception Class	149
Implementing the <code>InsufficientFundsException</code>	150
Defining the <code>InsufficientFundsException</code> Class	150
Modifying the Debit Method	151
Testing the <code>BankAccount</code> Class	152
Debugging a <code>VisualAge</code> for Java Program	153
Inspectors	153
The Debugger	156
Chapter 8. File I/O and Persistence	163
Persistence	164
Java I/O	164
Input Streams	165
Output Streams	166
Utility Classes	167
Using Input/Output Streams	168
Object Streams	170
Object Serialization	170
The <code>Serializable</code> Interface	171
Object Dependencies	172
Behind the Scene of Serialization	173
Controlling the Serialization	173
BankAccount Serialization	174
Serializing the Transaction Class	174
Serializing the <code>BankAccount</code> Class	177
Chapter 9. Managing Your Code	181
Storing Your Code	182
The Workspace	183
The Repository	183
Version Control	184
Editions	184
Versions	184
Versioning Program Elements	185
Managing Editions	187
Searching for Program Elements	191
The Search Dialog	191
Customizing the Search	191
Versioning the <code>BankAccount</code>	192

Part 2. Visual Building with JavaBeans

Chapter 10. Building User Interfaces with Java	197
The Abstract Windowing Toolkit	198
The GUI Components	198
Applets and Applications Revisited	201
Creating User Interfaces	203
Container Classes	203
Coding Your First User Interface	203
Handling Events	205
Handling Events with Java AWT 1.0	205
Handling Events with Java AWT 1.1	208
Controlling Layout	214
FlowLayout.	215
BorderLayout	216
CardLayout.	217
GridLayout	218
GridBagLayout.	221
Combining Layout Managers	224
Creating and Using Menus.	224
Using Menus with Applications	224
Using Menus with Applets	226
Chapter 11. Visual Programming and JavaBeans	229
Software Components	230
Software Components and Classes	230
Benefits of Software Components	230
JavaBeans	231
Visual Programming	231
The Visual Composition Editor	232
First Contact	233
Basic Bean Manipulations.	236
Creating a Simple User Interface	238
Visual Programming in Action	238
Looking at the Generated Code	241
More about JavaBeans	246
Bean Features	246
Introspection and BeanInfo Class.	249
Bean Persistence	250
Using VisualAge to Create Beans	251
Requirements for a Simple Calculator	251
Building the Calculator Model	252
Building the Calculator View	256
Connecting the Model and the View	257
Adding Menus to Your Application	270
Importing and Beanizing Existing Java Code	274

Chapter 12. Visual Programming in Action	277
Expanding the Banking Model	278
Creating an AtmCard Class	278
Modifying the BankAccount Class	281
Creating a Bank Bean	289
Creating the ATM View	292
Reusing Composite Beans	296
Organizing the Building of Your Beans	296
Using Layout Managers in Visual Composition	298
AtmAccountPanel	300
AtmKeypad	309
AtmKeypadPanel	313
AtmWelcomePanel	315
AtmPinPanel	319
AtmPanel	321
AtmHistoryPanel	327
Putting Your Applet Together	329
Assembling the ATM Panels	329
Displaying a Java Exception in a Message Box	336
Creating a Custom Event	338
Running Your ATM Applet As an Application	344
Converting Your Applet	344
The Final Touch	345
Using Factory Bean	346
Running CalculApp from AtmApplet	346

Part 3. Applet Publishing and Advanced Features

Chapter 13. Improving Your ATM Applet	351
Multithreading Support	352
The StaticTicker Bean	352
A Demanding Ticker	354
Multithreading to the Rescue	356
Modifying the ATM Applet	359
Resource Locking	360
NLS Support	361
Internationalization Framework	361
Locales	362
Obtaining All Supported Locales	362
Resource Bundles	364
Formatting Data	367
Collation	371
Character Properties	373
Converting Non-Unicode Text	375
Creating Multilingual Applications	375

Modifying the ATM Applet	375
Run-Time Type Identification	386
Using Class Objects	387
Identifying the Type of Account	387
Accessing Methods by Downcasting	389
Chapter 14. Publishing Your Applet.	393
Exporting Your Applet	393
Exporting the ATM Applet As Binary Files	394
Exporting the ATM Applet As a JAR File	395
Writing an HTML Page	396
Applets and the APPLET Tag	396
Referring to JAR Files in Your HTML Page	398
Sample of HTML Page for the ATM Applet	398
Setting Up Your Web Server	400
Hardware and Software Requirements	401
Installation	401
Starting the Server	403
Configuring the Server	404
Web Browsers and JDK 1.1	408
Sun HotJava 1.0	408
Netscape Communicator 4.03	409
Microsoft Internet Explorer 4.0	410
Compatibility Issues	410
Chapter 15. VisualAge for Java Enterprise Edition	413
Introduction to the Enterprise Access Builder	414
Relational Database Access	414
CICS Access	419
C++ Access	424
RMI Access	430
The ATM Applet Revisited	438
Accessing the Bank Database	438
Accessing the Credit Card Authentication Server	439
Accessing Distributed Java Objects	439
Home Banking	439
Appendix A. VisualAge for Java Installation	441
Installing VisualAge for Java	441
Loading the Samples in the Workspace	442
Managing the ATM Applet Resources	443
Loading IBM Extra Beans	443
Extra Software	444

Appendix B. UML Notation	445
Class	446
Object	446
Generalize/Inherits Relationship	447
Association Relationship	447
Aggregation Relationship	448
Sequence Diagram	449
Appendix C. Related Publications	451
International Technical Support Organization Publications	451
Redbooks on CD-ROMs	452
Other Publications	453
How To Get ITSO Redbooks	455
How IBM Employees Can Get ITSO Redbooks	455
How Customers Can Get ITSO Redbooks	456
Special Notices	457
Glossary	461
List of Abbreviations	477
Index	479

Figures

1. Quick Start Dialog	6
2. The Create Applet SmartGuide	7
3. The Workbench	8
4. Starting an Applet	8
5. Your First Applet	9
6. The Workbench Explained	10
7. The Workbench Tool Bar	11
8. Creating an Animated Applet	13
9. Creating the HiAgain Application	14
10. Defining the main() Method	14
11. The VisualAge for Java Console	15
12. Launching the Scrapbook in VisualAge for Java	36
13. Evaluating Java Code in the Scrapbook	36
14. Console Output of the For Loop Executed in the Scrapbook	37
15. Using Operators in the Scrapbook	38
16. Error Message in the Scrapbook Window	39
17. Object Representation	43
18. Car Object Representation	44
19. Object Sending Message	45
20. Description of The Bank Account Object	50
21. UML Representation of the BankAccount Class	51
22. The Complete BankAccount Class	56
23. The BankAccount Class and Some BankAccount Objects	57
24. Defining the Project, Package, and Class Name	61
25. BankAccount Class in the Class Browser Window	61
26. Creating a New Field	62
27. Creating the Getter Method	63
28. Code of Getter Method	64
29. Creating the setBalance method	64
30. Code of the Setter Method	65
31. Unresolved Problems Page: Incorrect Code	66
32. VisualAge for Java IDE Behavior Page	67
33. Code of Credit Method	68
34. Code of Debit Method	69
35. Console Output of the Test Program	70
36. Creating a ch04 Package in the Learn Java Project	76
37. Importing Packages with the SmartGuide	78
38. SmartGuide - Type of Import	80
39. Importing Classes	80
40. SmartGuide - Type of Export	81
41. Exporting Classes	82
42. Copying the BankAccount Class	96
43. The BankAccount Class in Package ch05	97
44. Defining Constructors in VisualAge for Java	98
45. Testing Your Constructor	99
46. Creating a Copy Constructor or a Finalizer	99
47. The Finalizer at Work	101
48. Inheritance Concepts	105

49.	The BankAccount Inheritance Hierarchy	105
50.	Invoking the Debit Method on Different Object Types	111
51.	UML Representation of Aggregation	115
52.	Creating the CheckingAccount class	123
53.	The CheckingAccount Class in the Workbench	124
54.	The SavingsAccount Class in the Workbench	125
55.	The OverdraftAccount Class in the Workbench	126
56.	Running the Subclasses Example	128
57.	Running the Abstract Class Example	130
58.	The Profitable Interface in the Workbench	131
59.	Selecting the Profitable Interface	132
60.	Running the ProfitableAccount Example	133
61.	Creating the Transaction Class	134
62.	The Transaction Class in the Workbench	135
63.	Running the Aggregation Example	138
64.	An Exception Condition	141
65.	The Java Exception Hierarchy	146
66.	InsufficientFundsException Constructor	151
67.	Running the Exception Example	153
68.	An Inspector Window	154
69.	Changing the Value of a Field	155
70.	Evaluating Code in the Context of an Object	156
71.	Breakpoint Set in the Debit Method	158
72.	Debugger Window	159
73.	Debugger Navigation Toolbar Buttons	160
74.	Changing a Value During a Debug Session	162
75.	Reading Account Identifier from Standard Input	169
76.	Output from Program	170
77.	The Serialization Process	172
78.	TransactionTest Output	176
79.	Restoring the BankAccount Objects.	179
80.	Managing your Code in VisualAge for Java	182
81.	Program Elements in the Workspace and Repository	185
82.	Versioning Program Elements	186
83.	Loading Editions	188
84.	Import SmartGuide	189
85.	The Repository Explorer	189
86.	Comparing Two Editions	190
87.	The Search Dialog	191
88.	Search Set Customization	192
89.	The Versioned Account Package	193
90.	The New Edition of the BankAccount Class	194
91.	AWT Class Hierarchy	199
92.	The MenuComponent and Subclasses	200
93.	Lifecycle of an Applet	202
94.	Placing Components in a Container	205
95.	FlowLayout In Action	215
96.	BorderLayout in Action	216
97.	CardLayout in Action	218

98. GridLayout in Action	220
99. GridBagLayout in Action	222
100. GridBagLayout Constraints	223
101. Frame with a Menubar	226
102. Applet with a Pop-up Menu	227
103. Visual Composition Editor	233
104. Class Browser Tabs	234
105. The Beans Palette	234
106. The Toolbar	235
107. The Free-Form Surface	235
108. The Add Bean... Dialog	237
109. Property Sheet of a TextField Bean	238
110. A Frame on the Free-Form Surface	239
111. A Frame with a TextField and a Button.	240
112. The Resulting Simple User Interface	241
113. Property Change Support	247
114. Calculator Model	251
115. Calculator View	252
116. Calculator BeanInfo Page	254
117. CalculApp Frame Window	257
118. Calculator Property Sheet	260
119. Selecting a Source Event for a Property-to-Property Connection	261
120. Property-to-Property Connections of the CalculApp	261
121. Calculator Bean Method and Property Features	262
122. CalculApp with All Connections	263
123. Script Dialog Window	265
124. CalculApp with the ColorEditor	267
125. Final Calculator Application	268
126. CalculApp with Menu Bar	272
127. CalculApp with Menu Bar and Pop-up Menu	274
128. AtmCard Class	279
129. BeanInfo Page of the BankAccount Class	285
130. BeanInfo Page for the Updated BankAccount Class	288
131. Bank Class	289
132. Accounts Property Creation	291
133. ATM Welcome Panel Sketch	293
134. ATM PIN Panel Sketch	294
135. ATM Balance Panel Sketch	295
136. ATM Balance and Transactions Panel Sketch	295
137. Reusing Visual Beans	296
138. Composite Bean Dependency Tree	297
139. Prompter Window	302
140. AtmAccountPanel	303
141. Tabbing Order in AtmAccountPanel	303
142. Changing Tabbing Order in Beans List	304
143. Connecting a BankAccount Variable to AccountTextField	305
144. Property-to-Property Connection Property Sheet	306
145. Tear-Off Property Dialog	307
146. Populating CardNumber in CardTextField	308

147. Promoting a Feature	309
148. AtmKeypad	310
149. Event-to-Script Connection - Properties Window	312
150. AtmKeypad with Script Connections	312
151. Visual Feedback from a BorderLayout	313
152. AtmKeypadPanel	314
153. Promoting the EchoChar Property Feature	315
154. AtmWelcomePanel	316
155. AtmWelcomePanel Layout	316
156. AtmWelcomePanel with Connections	319
157. AtmPinPanel	319
158. AtmPanel	322
159. AtmPanel with Connections	326
160. Reorder Connections Window	327
161. AtmHistoryPanel	327
162. Switching Page with a CardLayout Manager	330
163. Using Connections to Switch Cards	332
164. Switching Cards from AtmPinPanel	333
165. Switching Cards from AtmPanel	334
166. Switching Cards from AtmHistoryPanel	335
167. Displaying a Java Exception in an IMessageBox Bean	337
168. InsufficientFundsException Displayed in a Message Box	338
169. Check Pin Number Event-Trace Diagram	340
170. SmartGuide - New Event Listener Window	341
171. SmartGuide - Event Listener Methods Window	341
172. Enabling PIN Checking with Connections	344
173. Running AtmApplet As an Application	345
174. StaticTicker Bean in Action	354
175. Multithreaded Ticker in Action	359
176. AtmApplet with the Ticker Bean	360
177. LocaleListApplet in Action	363
178. ATMLanguagePanel	379
179. AtmWelcomePanel with AtmLanguagePanel	379
180. NLS Version of AtmWelcomePanel in Action	386
181. Identifying the Class Referenced by the BankAccountVariable	389
182. Downcasting a BankAccount Object	390
183. ATM Applet Running in the HotJava Browser	400
184. Selecting Components	402
185. Selecting Directories for Your Components	402
186. Confirmation Message Box	403
187. Invoking a CICS Transaction from a Java Applet	421
188. Java to C++ Layering	426
189. Relationship between the Source Code and the Generated Code	429
190. Client-Server Interactions through the Proxy Bean	438
191. UML Class Notation	446
192. UML Object Notation	446
193. UML Generalization/Inheritance Relationship Notation	447
194. UML Association Relationship Notation	448
195. UML Aggregation Relationship Notation	448
196. UML Sequence Diagram Notation	449

Tables

1. Toolbar Icons	11
2. VisualAge for Java Symbols	16
3. Primitive Data Types.....	23
4. Access Possibilities	72
5. VisualAge for Java Projects and Their Contents.....	82
6. Components Methods	199
7. Deprecated Methods	207
8. Property-to-Property Connection Results	258
9. AtmAccountPanel GridBagLayout Constraints.....	302
10. AtmWelcomPanel GridBagLayout Constraints.....	317
11. AtmPinPanel GridBagLayout Constraints	321
12. BalancePanel GridBagLayout Constraints	324
13. OperationPanel GridBagLayout Constraints	324
14. AtmHistoryPanel GridBagLayout Constraints	328
15. String Comparison Strengths	372
16. DateTimePanel GridBagLayout Constraints.....	377
17. LanguagePanel GridBagLayout Constraints.....	378
18. Data Access Class Library Reference.....	417
19. Classes Generated by Data Access Builder	418

Part 1



VisualAge for Java and Java Basics

Welcome to Programming with VisualAge for Java. This part begins with a short introduction to the VisualAge for Java development environment, so you can build your first applet and application quickly. You then start building your Java skills by learning the essential elements of the language. Next, you learn the terminology and concepts used in object-oriented technology, followed by Java packages, exceptions, persistence, and code management.

In each chapter, we illustrate the newly learned concepts with a bank account and automated teller machine (ATM) example, which you enhance as you proceed to more advanced features of the Java language and VisualAge for Java environment.

1

Introduction to the Environment

This chapter presents an overview of the VisualAge for Java development environment. You learn the basic terms that you need to understand to create your first program. Before you finish reading this chapter, you will have your first Java program up and running!

The VisualAge for Java Development Environment

VisualAge for Java is an integrated, visual environment that supports the complete cycle of Java program development. With the VisualAge for Java programming environment, you can build 100% pure Java applications, applets, and JavaBean components that run on any Java-compatible Virtual Machine (VM) Java Development Kit (JDK 1.1) or inside any JDK 1.1-enabled browser. Using the true rapid application development (RAD) provided by VisualAge for Java, you can shorten the development life cycle of your applications and improve their time to market.

VisualAge for Java features include:

- **An integrated development environment (IDE)**, which enables you to create classes, or change methods, and then incrementally compile without the need to exit the testing phase of development. This IDE sports:
 - A fully featured colored-syntax editor, which helps you write syntax-free source code. The source code is incrementally compiled when you save it.
 - A debugger, which opens automatically when program exceptions occur
 - Single-user version control, which lets you keep track of all changes you make to your code
 - A Visual Composition Editor, which allows you to develop your application visually
 - A JavaBeans creation tool to create 100% pure Java beans that you can use with the Visual Composition Editor
- **Enterprise Access Builders** (IBM VisualAge for Java Enterprise version only), which include:
 - Database Access Builder for accessing enterprise data managed by database servers such as IBM's DB2 using the Java Database Connection (JDBC) specification
 - CICS Access Builder for building CICS Java external call interfaces (ECIs) to enterprise transactions managed by the IBM CICS Transaction Server for OS/390
 - Distributed Application Builder for building remote method invocation (RMI) or C++ interfaces that connect to C++ and Java applications running on a server

Also planned for the Enterprise product is a fully integrated, repository-based team environment that allows management of the development process on Java projects with complete source and version control.

The rest of this book introduces you to the Java language and object-oriented (OO) concepts. You will be using the Professional Edition of VisualAge for Java which does not include the Enterprise Access Builders and the team environment support.

Building Your First Applet

The purpose of this chapter is to get you started quickly with VisualAge for Java. Before starting, you should have a look at the following table of terms used in OO technology. These terms are explained later in the book, but it helps to have a rough idea now of what they mean.

Terms Used in Object Oriented Programming in Java	
Class	A template for creating objects. A class defines the behavior and properties that are common to all objects of that class.
Object	An instance of a class. An object shares the behavior of all objects of the same class, but each object can have a different state.
Applet	A special kind of class introduced in Java. Its instances usually run in a Web browser such as Netscape Navigator.
Methods and messages	Objects communicate with each other by sending messages. When an object receives a message, a corresponding method, defined in the class definition, is invoked to perform the required task.
Project	The basic Java environments provide only the concept of a package to organize your work. In VisualAge for Java you also have projects, within which you can organize your packages.
Package	A collection of classes that serve a common purpose. This is Java's way of organizing classes into logical entities that are easier to maintain and understand than a huge set of classes all at the same level.

Let's Get Started!

Before you go any further, you must have VisualAge for Java installed on your computer. If you have not already installed the product, refer to Appendix A on page 441. Make sure you also read the installation instructions included with the CD-ROM that accompanies this book for the latest changes.

Our first Java class is a simple applet that displays the text of your choice in the applet's window. You launch VisualAge for Java by double-clicking the IBM VisualAge for Java icon in the IBM VisualAge for Java folder. The **Quick Start** dialog (Figure 1) is displayed.

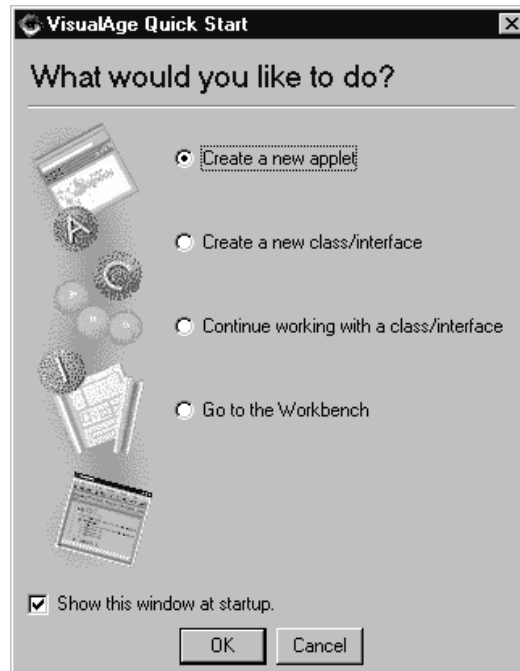


Figure 1. Quick Start Dialog

The Quick Start dialog is shown when you start the VisualAge for Java IDE, unless you deselect the checkbox at the bottom of the window.

Another window opened during startup is the Workbench, where you usually create and manipulate your classes. From the Workbench you can also launch the Quick Start window, by selecting **Window→Quick Start** from the menu bar.

Using the Quick Start dialog you can create a new class or continue working with an existing class that you created previously. Because this is the first time you are using VisualAge for Java, you do not have anything to continue working with, so the default selection (**Create a new applet**) is fine.

SmartGuides

VisualAge for Java makes use of **SmartGuides** (you may be familiar with similar function called Wizards in other products) throughout the product to help you create Java applets and applications. Clicking the **OK** button on the Quick Start window opens the **Create Applet** SmartGuide (Figure 2), which guides you through the process of creating your new applet class.

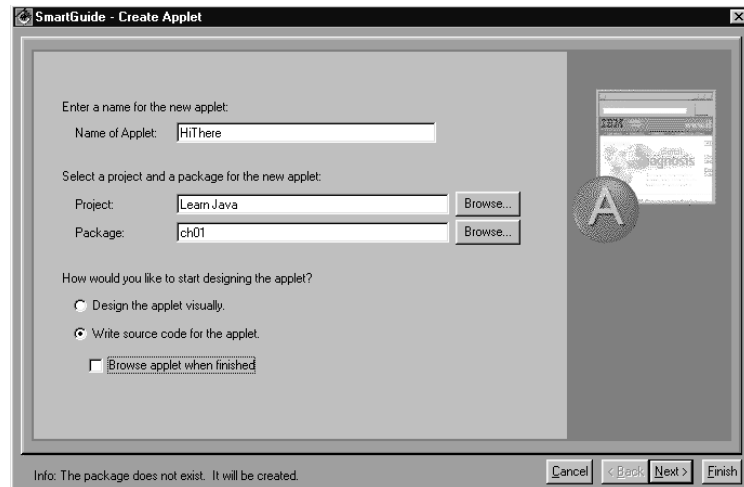


Figure 2. The Create Applet SmartGuide

To create your applet, fill in the text fields (Name of Applet, Project, and Package) as shown in Figure 2. Select the **Write source code for the applet** radio button and deselect the **Browse applet when finished** checkbox. Click the **Finish** button to complete your work with the SmartGuide window. VisualAge for Java now creates the code needed for your new applet. The dialog closes and lets you work with the Workbench window (Figure 3).

Believe it or not, you have just created your first applet using VisualAge for Java! Now you can examine and test your applet, using the WorkBench.

The Workbench

The Workbench is the main control center of the VisualAge for Java IDE. From the Workbench, you can access projects, packages, and classes and modify and test your code with just a few mouse clicks.

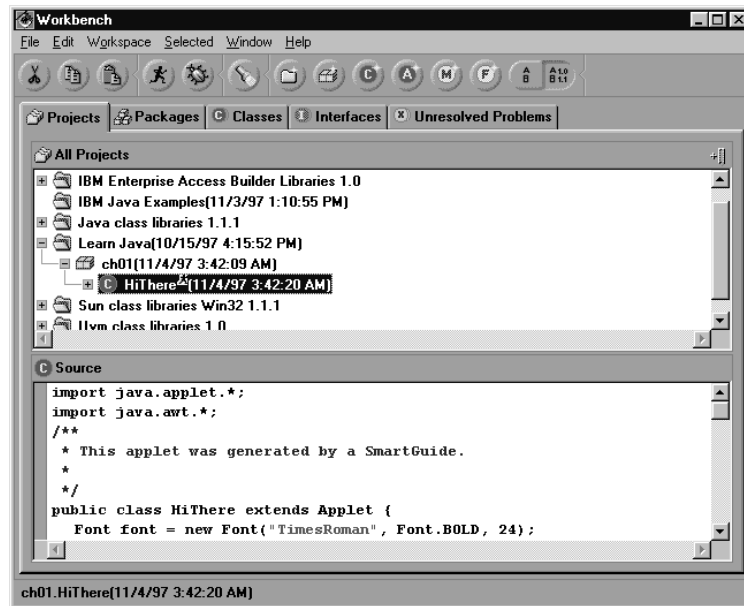


Figure 3. The Workbench

To test your newly created applet, expand your project and your package by clicking the plus sign to the left of them and select the **HiThere** applet by clicking it. Select **Selected**→**Run**→**In Applet Viewer** from the menu bar of the Workbench; VisualAge for Java prompts you for possible parameters to use when the applet is started (see Figure 4). You can also change the initial width and height of your applet. In this case the defaults are just fine, and you can click the **Run** button.

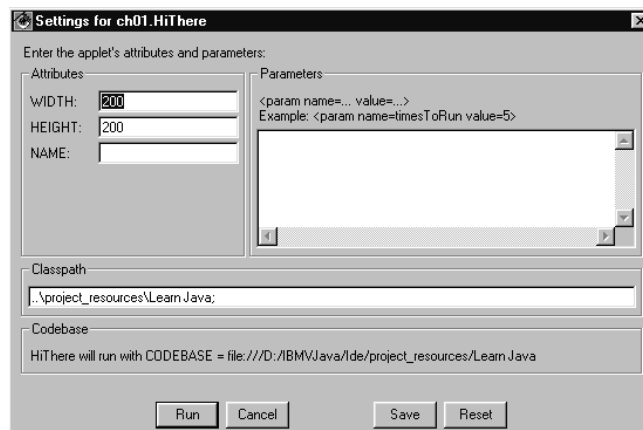


Figure 4. Starting an Applet

After you click the **Run** button, the Applet Viewer opens and runs your applet (Figure 5). The Applet Viewer is a utility that lets you test your applets without having to use a Web browser. You may have to resize the window to see the complete message.



Figure 5. Your First Applet

Congratulations, you have built and tested your first applet with VisualAge for Java!

Modifying Your Applet

To modify the applet, select your class in the Workbench. The class definition of the applet is displayed in the Source pane of the Workbench window and looks like this:

```
public class HiThere extends java.applet.Applet {
    java.awt.Font font = new java.awt.Font("TimesRoman",
        java.awt.Font.BOLD, 24);
    java.lang.String str = "Welcome to VisualAge";
    int xPos = 5;
}
```

To change the text “Welcome to VisualAge” to any string you like, just overwrite the text. Save your changes by selecting **Edit→Save** from the menu bar. VisualAge for Java compiles the code immediately, and you can test the result by again selecting **Selected→Run→In Applet Viewer** from the menu bar. If you did not close the Applet Viewer window you started while testing the previous version of the applet, you can also select **Applet→re-start** from the menu bar of the running Applet Viewer.

This simple example shows how VisualAge for Java can help you create Java programs. You did not need to edit, save or compile any file. You simply changed the code generated automatically by VisualAge for Java, saved it, and ran it. This reduced development life cycle is a reality thanks to the incremental compiler, which compiles your code on the fly when you save it.

The Workbench Explained

You will probably spend a lot of time using the Workbench, so it's time to take a look at the Workbench (Figure 6) and its features.

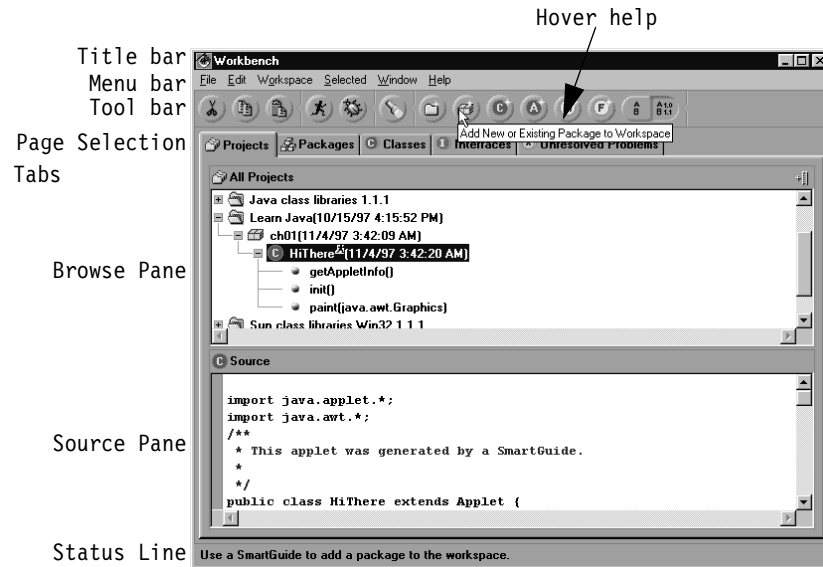


Figure 6. The Workbench Explained

The Workbench contains two panes: the Browse pane in the upper part and the Source pane in the lower part. The contents of the Browse pane depend on the current tab selection above the pane. To switch between the *Projects*, *Packages*, *Classes*, *Interfaces*, and *Unresolved Problems* views, just click the corresponding tab. A tree-view of the projects, packages, classes and interfaces, which are currently loaded in your workspace, is shown in the Browse pane when you select the *Projects*, *Packages*, *Classes* or *Interfaces* views. The *Projects* view is the default view shown when you open the Workbench for the first time. To collapse or expand any item in the list, click the plus or minus sign to the left of the item. To show the source code for an item in the Source pane, just select the item, using your mouse.

Some of the terms in the previous paragraph are probably new to you: not to worry, they are explained as you work through the book.

From the Workbench you can access the menu of the currently selected item (project, package, class, interface, or method) in two different ways: you can access the pop-up menu by right-clicking the selected item, or you can use the menu bar the way you used it to test your first applet. The pop-up menu of the Source pane and

the Edit menu in the menu bar have the same contents. The Browse pane's pop-up menu is the same as the Selected menu in the menu bar.

Figure 7 shows the tool bar icons described in Table 1.

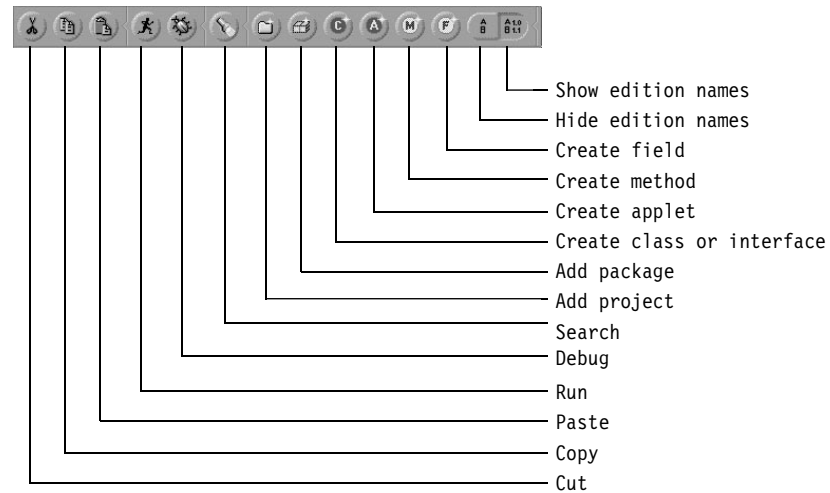


Figure 7. The Workbench Tool Bar

Table 1. Toolbar Icons

Icon	Description
Cut	Move the currently selected text to the clipboard.
Copy	Copy the currently selected text to the clipboard.
Paste	Copy the contents of the clipboard to the current cursor position.
Run	Run the selected class or a class from the selected project or package.
Debug	Start the debugger.
Search	Search for a reference or declaration of a field, method, or class.
Add Project	Add a project to the current workspace.
Add Package	Add a package to the current select project.

Table 1. Toolbar Icons

Icon	Description
Create Class or Interface	Invoke the Create a new Class or Interface SmartGuide.
Create Applet	Invoke the Create a new Applet SmartGuide.
Create Method	Invoke the Create a new Method SmartGuide on an existing class.
Create Field	Invoke the Create a new Field SmartGuide on an existing class.
Hide Edition Names	Hide the edition names of program elements.
Show Edition Names	Show the edition names of program elements.

Tip

You can expand the Source pane or the Browse pane to fill the whole window by double-clicking the title area of the pane. Another double-click on the title area restores the previous split view.

You can also change the relative size of the two panes by dragging the split line between the panes with your mouse.

The Source pane can also be used to add a comment to a project or a package.

Creating An Animated Applet

Now that you are more familiar with the VisualAge for Java IDE, it is time to create your second applet. With VisualAge for Java, it is easy to create applets that scroll text from one side of the applet to another.

This time you create a new class without Quick Start. Instead, click on the **Create Applet** icon on the tool bar of the Workbench. The SmartGuide appears again, requesting you to fill in information about your second applet. Use the same project and package names as in your first applet, and name your applet *HiThere-Again*. Select the **Write source code for the applet** radio button

and deselect the **Browse applet when finished** checkbox. This time, click the **Next>** button to access the second page of the SmartGuide (Figure 8).

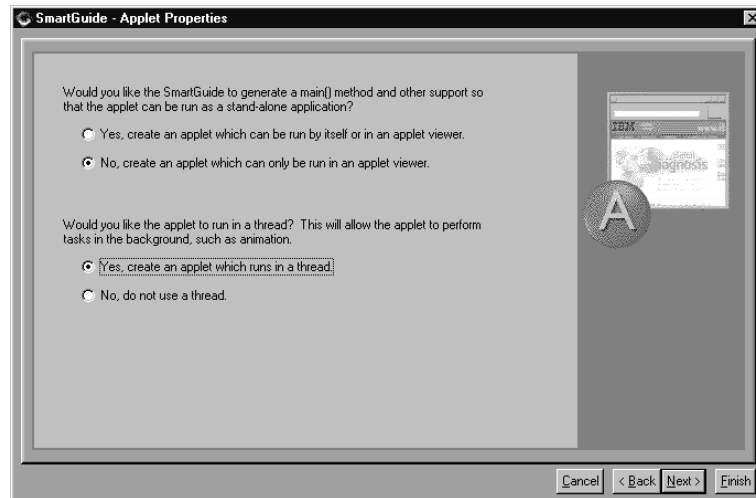


Figure 8. Creating an Animated Applet

Select the **Yes, create an applet which runs in a thread** radio button, and click the **Finish** button. Run the applet by selecting it in the Workbench and then selecting **Selected→Run→In Applet Viewer** from the menu bar.

Your application should show a marquee text scrolling from left to right. You have just built an animated applet.

Building Your First Application

In this section you create a Java *application* that prints a string to the VisualAge for Java console. The console is a window that displays messages sent by your application to the standard output of the operating system. To create a class that can be run as an application, without the Applet Viewer or a Web browser, you have to implement a method called `main` in your class.

In the Workbench, select your package (ch01), and select **Selected→New Class/Interface** from the menu bar. Enter *HiAgain* for the Class Name. Notice that because you selected the ch01 package within the Learn Java project, you do not have to type them in again (Figure 9).

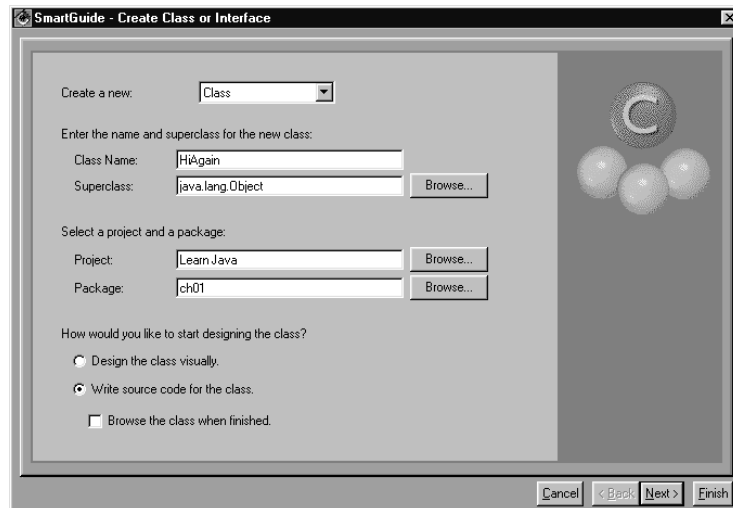


Figure 9. Creating the HiAgain Application

Now click the **Next>** button to access the second page of the SmartGuide, which lets you specify attributes of your class.

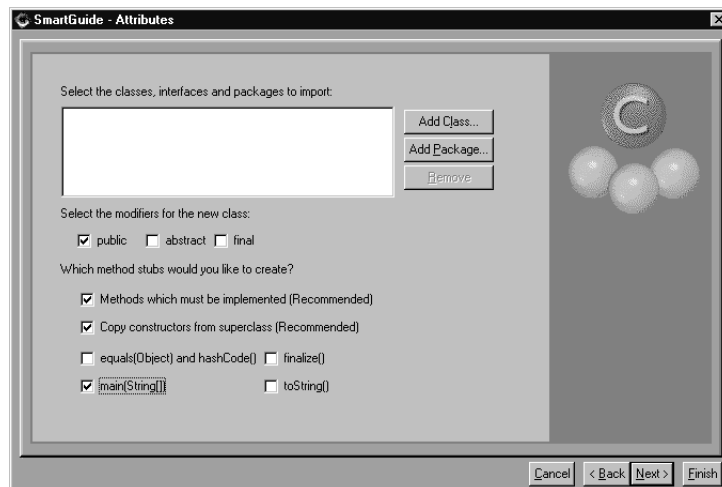


Figure 10. Defining the main() Method

Select the **main(String [])** checkbox and click the **Finish** button.

In the Source pane of the Workbench you can see the definition of your newly created class. In the Browse pane, expand the class by clicking the plus sign to the left of it, and you can see the *main* method. Select this method, and the Source pane shows you the

method implementation. Only a stub of the method has been generated, but you can change that by adding the following code to the method body:

```
System.out.println("This is my first application!");
```

The `System.out.println()` statement prints a string to the standard output, which in turn is displayed in the VisualAge for Java Console window.

Save any changes you made, using the menu bar (**Edit**→**Save**), and you are ready to see the results of your work. Select the class and from the menu bar select **Selected**→**Run**→**Run Main**. VisualAge for Java asks you for parameters, but you do not need to specify any now, so just click the **Run** button. The Console window (Figure 11) opens to display the result.

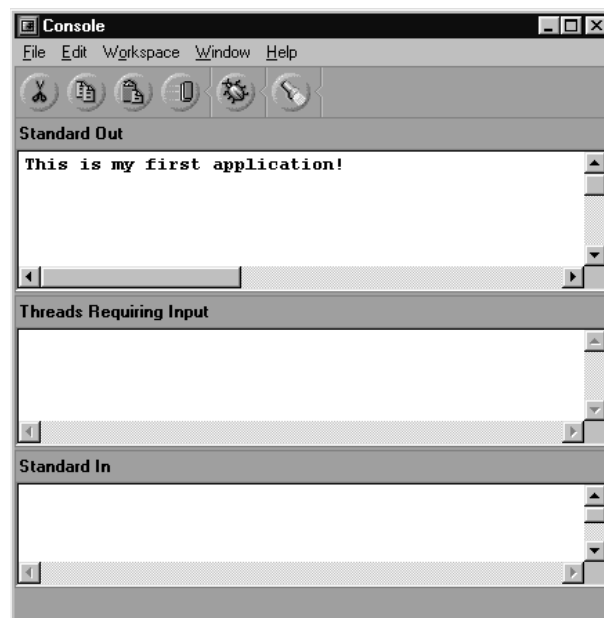


Figure 11. The VisualAge for Java Console

You have successfully created another Java class. The Console window that displays the text that your application generated is used as a standard output window for messages and to enter input through the standard input. VisualAge for Java does not have a command line that is used for the same purpose in standard command-based Java environments such as JDK 1.1 from Sun Microsystems.

Tip

With VisualAge for Java, you can easily build an applet that can also be run as an application. To do so, your applet has to implement the `main()` method to handle opening a window without an Applet Viewer or a Web browser.

To create this kind of applet, just click the Applet icon in the tool bar, and the SmartGuide creates the necessary implementation for you. After providing names for the project, package, and class, select the **Write source code for the applet** radio button and click the **Next>** button to access the Applet Properties SmartGuide. Select the **Yes, create an Applet which can be run by itself or in an Applet viewer** radio button, and click the **Finish** button. Notice that the **Select-ed→Run** menu has both **Run main** and **In Applet Viewer** options available if you select the class in the Workbench.

VisualAge for Java Symbols Explained

As you probably have already noticed, VisualAge for Java uses different colors and symbols to represent different types of classes and methods. Once you learn these symbols, it is very easy to see at a glance which methods your classes have. Table 2 lists the symbols that VisualAge for Java uses to visualize different elements.

Table 2. VisualAge for Java Symbols

Symbol	Name	Description
	Project	A project folder that contains packages. This icon appears either opened (expanded) or closed.
	Package	A package that contains classes and interfaces
	Class	The Workbench displays this icon before the name of each class definition.

Table 2. VisualAge for Java Symbols

Symbol	Name	Description
	Interface	The Workbench displays this icon before the name of each interface definition.
	Executable	Shows whether a class is executable as either an applet or an application
	Abstract	Displayed when a class or method is defined as abstract
	Public	Green circle: public access modifier
	Default	Blue triangle: default access modifier
	Private	Red square: a private access modifier
	Protected	Yellow diamond: protected access modifier
	Native	A native method
	Transient	A transient field
	Synchronized	A clock symbol: a synchronized method
	Final	A final method
	Static	A static method
	Visual Class	The class has methods generated by the VisualAge for Java Visual Composition Editor.
	Generated	The method was generated by VisualAge for Java and should not be modified.

Table 2. VisualAge for Java Symbols

Symbol	Name	Description
✖	Error	A red cross can be shown in front of methods or classes. A red cross in front of a method indicates the method has errors, and the class in question has a grey cross. A red cross in front of a class indicates that the class definition has errors.

Summary

VisualAge for Java provides you with an integrated environment, where you can quickly and easily develop Java classes.

In this chapter you have created both a Java applet and application with a few mouse clicks.

The tool set includes several tools, such as the Workbench to browse and edit your projects, packages and classes; SmartGuides, a set of wizards, to generate initial code; and the Console window.

2

Java Basics

This chapter is divided into two sections. The first section contains information about how Java works and introduces the components of the Java language: basic data types, statements, variables, operators, and control flow. The second section shows how to use VisualAge for Java to reinforce your new Java knowledge.

How Java Works

Java is more than just the Java language. It also includes the Java Virtual Machine (VM), which is like a computer implemented in software. The Java VM is capable of executing instructions called Java *bytecodes*. Whenever you save any changes you make to source code, VisualAge for Java compiles the source code into bytecodes. VisualAge for Java incorporates its own VM that is capable of executing bytecodes. The Java VM is the key to a Java program's portability.

If you use a Java-enabled Web browser to connect to a Web page that contains an applet, the browser's VM is responsible for executing the bytecodes of that applet. Therefore as long as a Web browser implements or supports a Java VM, it does not matter on which platform the browser runs. This is the key to machine independence.

The Java VM has another advantage: because Java VMs have their own instruction set, it is difficult to create applets that can damage your computer when executing in your browser. The Java VM also includes a bytecode verifier that inspects the bytecodes before they are actually run. Thus undesired bytecodes that could be harmful cannot execute. In addition, applets that are downloaded from the Web cannot read and write access to files on your hard disk by default. The applets running in your browser are in a protected environment, called a *sand-box*, that is well isolated from your computer.

The Java Language

Consider this chapter a crash course in the Java language, not a reference manual. If you are already familiar with a language like C++, you will easily recognize what is new or different. Here are some of Java's most important features:

Simplicity	Java is an attempt to simplify C++ while retaining enough expressive power to be useful. It does not add lots of syntactic sugar or unnecessary features.
Object-orientation	Almost everything in Java is either a class, a method, or an object. Only the most basic data types (for example, int) are primitives.
Portability	Java programs are compiled to a bytecode format that can be run by interpreters on many platforms including Windows 95, Windows NT, Solaris, AIX, and OS/2.
Safety	Java code can be executed in an environment that prohibits introducing viruses, deleting or modifying files, or otherwise performing data-destroying and computer-crashing operations.
Performance	Java can be compiled to native code on the fly.

Multithreaded Threading support is built into the language. A single Java program can have many different tasks processing independently and continuously.

Java's popularity is due to the fact that it is a great language for programming applets which can be downloaded from the Web. However, Java is a full-blown language that is not restricted to programming applets. You can program almost everything in Java that you can program in C++ or Smalltalk.

Statements

The source code for a Java program is made up of statements. Statements include:

- ❑ **Expressions:** combinations of keywords, operators, and variables.
- ❑ **Blocks:** sets of statements enclosed in curly braces: {}.

You can also include comments and white space in Java programs; the compiler ignores both.

Variables, Keywords, and Operators

In Java, as in most programming languages, variables, keywords, and operators are the building blocks of expressions; they are the basic vocabulary of the language.

Variables

Variables in Java are declared with the following format:

```
[variable modifiers] variabletype variablename [= initialValue];
```

Instance or class variables must be declared within the class declaration. Local variables can be declared anywhere within a method.

Examples of declarations are:

```
int amount; // integer value named amount: initial value 0 if class
            // or instance variable, otherwise you get a compiler
            //error
int amount = 1000; // as above, but assigns an initial value
                  // of 1000
private int amount = 1000; // as above but with limited access
```

Modifiers

The variable modifiers specify information about the accessibility and persistence of a variable. The valid values are:

public	The variable is fully accessible from anywhere in your program.
protected	The variable is accessible in the package defining the class and within the body of any subclass. Packages and subclasses are fully explained in Chapter 4, “Organizing Your Code,” on page 71.
private	The variable is accessible only within objects of the class within which it is defined.
static	Static variables belong to the class, not to instances, and are sometimes referred to as <i>class variables</i> . Variables that are not static are called <i>instance variables</i> .
final	The value of the variable cannot be changed once it is set. It is effectively a constant.
transient	The variable is not part of the persistent state of an object.
volatile	The variable may be modified asynchronously.

If a variable access modifier (public, protected, or private) is not selected, the default is *package* access. This are introduced in Chapter 4, “Organizing Your Code,” on page 71.

Variable Types

Java is a strongly typed language. Therefore every variable must have a declared type. The type of a variable determines which values it can take and which operations are allowed on it. The primitive Java types include boolean, char, byte, short, int, long, float, and double.

Table 3. Primitive Data Types

	Width in Bits	Initial Value Set by Compiler	Possible Values
boolean	8	false	true or false
char	16	'\u0000'	Unicode character from 0 .. 65535
byte	8	0	-128 .. 127
short	16	0	-32768 .. 32767
int	32	0	-2^{31} .. 2^{31}
long	64	0	roughly $\pm 9.22\text{E}+18$
float	32	0.0F	roughly $1.4\text{E}-45$ to $3.40\text{E}+38$
double	64	0.0D	roughly $4.9\text{E}-324$ to $1.7\text{E}+-308$

Java also supports nonprimitive or user-defined types: these are *arrays*, *classes*, and *interfaces*. Arrays are explained in “Predefined Classes: String and Array” on page 29. Classes and interfaces are introduced in Chapter 3, “Objects and the Java Language,” on page 41 and Chapter 6, “Reuse in Java,” on page 103.

Variable Names

Variable names, or identifiers, in the Java language must start with a letter, underscore, or dollar sign. The valid characters are:

- ❑ Characters **A** through **Z**
- ❑ Characters **a** through **z**
- ❑ Unicode characters above **hex 00C0**

Subsequent characters can also be digits.

Initial Values

Within a class declaration, a variable can have an initial value. If you omit the initial value, the variable gets a default value as defined in Table 3. If the variable is a local variable (not part of a

class declaration), an initial value is not assigned, and the compiler may report an error if you do not explicitly assign an initial value.

Literals

Literals are the actual values you can assign to some types of variables. There are five types of literals:

- ❑ **Integer literals** of type `int`, such as 123 unless they are larger than 32 bits, or suffixed with L (for example, 123L), in which case they are `long`.
- ❑ **Floating point literals**, such as 3.14159 or 5F or 6D. The default is `double`.
- ❑ **Boolean literals** `true` and `false`.
- ❑ **Character literals**, such as 'a' or 'B' or `\r` (carriage return)
- ❑ **String literals** such as "this is a string."

Casting

You use casting to convert the current type of an object or primitive to another type. Consider the following code fragment:

```
double aDouble = 5.1;
int    anInt    = 0;

anInt = aDouble;    // incompatible types
```

The compiler complains if you try to assign the value of another type to an incompatible type as shown in the previous example. To satisfy the compiler you can use casting by telling the compiler it should convert the double value to an `int` type:

```
double aDouble = 5.123;
int    anInt    = 0;

anInt = (int)aDouble;    // compatibility enforced by casting
```

However, in this example you lose significant digits because you are converting from a larger to a smaller type. The value of `anInt` would be 5. It is your responsibility to use casting carefully.

This code example converts a value of a smaller type to a larger type:

```
double aDouble = 0;
int    anInt    = 10;

aDouble = anInt;    // casting not required
```

Explicitly casting in this example is not necessary because Java automatically converts a smaller type to a larger type.

Scope of a Variable

A variable's scope is the block of code within which the variable is accessible. The scope of a variable determines when the variable is created and when it is destroyed. The scope of a variable is established when the variable is declared. The scope of a variable varies according to the variable category:

- ❑ **Member variable:** The variable is a member of a class and is declared within a class.
- ❑ **Local variable:** The variable is declared within a method or within a block of code in a method. In this case, the local variable is accessible from its declaration to the end of the code block in which it was declared.
- ❑ **Method parameter:** The variable is a formal argument to methods and constructors. The scope of a method parameter is the entire method or constructor for which it is a parameter.
- ❑ **Exception handler parameter:** The variable is similar to the method parameter but is a parameter to an exception handler rather than a method or constructor.

In the rest of this chapter you will deal only with local variables.

Information



Java uses the Unicode character set, a standardized character-coding system to support Roman as well as non-Roman alphabets such as Japanese. In contrast to other formats such as ASCII, Unicode allows the encoding of up to 34,168 characters. In Java all identifiers, strings, and characters are composed of a series of Unicode characters.

The `java.io` package provides methods that use other encoding schemes to read and write characters to files using so that the files can be browsed and edited with traditional ASCII-based tools.

Keywords

Keywords are reserved words, that is they cannot be used in Java programs as variable names. There are about 50 keywords, including the following, which will be familiar to C and C++ programmers:

break, case, char, do, else, for, if, int, return, switch, void, and while;

and others like:

abstract, extends, false, true, implements, native, null, super, and synchronized.

Operators

As in any other programming language, Java supports different operators. In this section we outline each operator category and the operators that belong to it.

Unary Operators

-	negation: change the sign of the variable
~	bitwise complement: toggle each bit to 0 or 1
++	increment: increase the variable by 1
--	decrement: decrease the variable by 1

Binary Operators

+	addition: add two variables
-	subtraction: subtract one variable from another
*	multiplication: multiply one variable by another
/	division (truncates when used on integer values): divide one variable by another
%	modulus: take the remainder of dividing one integer by another

&	bitwise AND: perform a logical AND of each bit in the binary representation of the variable
	bitwise OR: perform a logical OR of each bit in the binary representation of the variable.
^	bitwise XOR: perform a logical exclusive OR of each bit in the binary representation of the variable
<<	left shift: shift all bits one position to the left (equivalent to multiplying by 2)
>>	right shift: shift all bits one position to the right (equivalent to dividing by 2)
>>>	unsigned right shift (fills top bits with 0)

Information

Binary operators may be combined with the assignment operator, as in the C language. For example:

`a -= b;` is equivalent to: `a = a - b;`

You cannot overload operators as in C++; that is, you cannot define operators on new types.

Integer and Boolean Relational Operators

!	not: toggle the logical value of the operand (can be combined with < or > or =)
>	greater than (can be combined with =)
<	less than (can be combined with =)
==	equality
&&	boolean AND: true if both operands are true, otherwise false
	boolean OR: true if either or both operands are true, otherwise false

Ternary Conditional Operator

?	as in C and C++, an abbreviated if-else operator
---	--

Member Access Operators

.	dereference
[]	access to array elements

Separators

;	defines the end of a statement
()	group expressions in method calls
{ }	define block of statements

Precedence of Operators

Expressions are evaluated from left to right. Within that order, operators with a higher precedence get evaluated first. Consider the following example:

```
int result;  
result = 10 + 6 / 3;
```

This would result in 12 because the / operator has a higher precedence than the + operator. The result of the following code example is 5. The addition (16) is performed first and then divided by 3. The integer division truncates the result to 5.

```
int result;  
result = (10 + 6) / 3;
```

Here is a list of operators and their precedence. Operators higher in the list have higher precedence.

```
[] . ()  
! ~ ++ --  
* / %  
+ -  
<< >> >>>  
< > <= >= instanceof  
== !=  
&  
^  
|  
&&  
||  
?:  
= operand=
```

You may want to use parentheses to indicate explicitly which expressions have to be evaluated first. The use of parentheses can help make your code easier to read and maintain.

Comments

The compiler ignores comments and white space (such as blanks, tabs, and line feeds or carriage returns). There are three types of comments:

```
/* This is a C-like comment, which can, if necessary,
   span several lines.
*/
// C++ or ANSI C type comment, which ends at the end of the line
/** Used for documentation. It is automatically converted
    into HTML, using the javadoc tool, which is a part
    of the JDK
*/
```

Note that string literals may not extend over one line; that is, carriage returns or line feeds are not ignored within a string literal.

Predefined Classes: String and Array

Strings and arrays are important language tools and it is important to be able to use them. It is not important to understand classes at this point, you can just think of Strings and arrays as other types that Java supports.

Strings

Java provides a class called *String*, which represents a sequence of characters, such as “Hello World.” Here are some examples of how you can manipulate strings using the String methods:

```
String s1, s2; // declare two string variables
int i; // declare an integer index
s2 = new String("this is a string"); // create a string
/*
   You could also just use: s2 = "this is a string"
*/

s1 = s2.toUpperCase();
if (s2.equals(s1))
    System.out.println("s2 is in upper case");

i = s1.length(); // sets i to 16
s2 = String.valueOf(i); // sets s2 to "16"
s1 = s2.concat("abc6"); // sets s1 to "16abc6"
```

```
s2 = s1.substring(0,3); // sets s2 to "16a"
i  = s1.indexOf("6",2); // look for the character 6, starting
                        // at index of 2 (sets i to 5)
```

Read This

The string equals operation or method tests the content of two strings for equality. By contrast, the == operator tests the value of two variables for equality. For a variable referring to an object, its value is that object reference, and two such variables are equal only if they are pointing to the same object. Thus:

```
s1 = "ab"; s2 = "a" + "b";
if (s1.equals(s2)); //true same contents
if (s1 == s2);      //false 2 different objects
```

StringBuffer

Java strings are immutable; that is, once they are created, they cannot be changed. Another class, *StringBuffer*, is a string of variable size. It is mainly used to create strings. The compiler uses it to implement the + operator. For example:

```
"a" + "b"
```

is compiled to:

```
new StringBuffer().append("a").append("b").toString();
```

Let's analyze this piece of code. The order of the operations is from left to right, and the operations are separated by a dot:

- | | |
|-----------------------|--|
| 1. new StringBuffer() | A new empty string buffer is created. |
| 2. append("a") | a is appended. |
| 3. append("b") | b is appended. |
| 4. toString() | The string buffer is converted to a string type. |

As expected, the result is a string with the value of ab.

Information

The *toString* method is used for converting any object to a string. The default *toString* returns the class name of the object and the address in the Java VM. Many classes create their own *toString* method to provide a useful description of themselves.

Arrays

Arrays are used to store a known number of objects of the same type. The declaration of an array variable contains the data type and the array name. The declaration format is:

```
type[] name;
```

To assign the variable to a new array use:

```
name = new type[size].
```

An example of an integer array is:

```
int i[];  
i = new int[10];
```

The example creates an array of 10 integers with a zero-based index. The indices run from 0 to 9.

Rather than using multidimensional arrays (which do not exist in the language) you use arrays of arrays:

```
int j[][] = new int[3][4]; // create a new integer array of  
                          // of 3 rows and 4 columns
```

For safety purposes, array subscripts (row and column numbers) are checked to make sure they are integers and valid. If not, a runtime error or *exception* occurs. In this case an *ArrayIndexOutOfBoundsException* is thrown. Exceptions are introduced in Chapter 7, “Error Handling and Debugging”.

You can determine the number of elements in an array by using the length operator:

```
int i[] = new int[5];  
int j;  
j = i.length;  
  
int matrix[][] = new int[3][4];  
int row0 = matrix[0].length;
```

You can create arrays of any type. Array dimensions are not dynamic; once an array has been created, its dimensions are fixed.

Control Flow

You use conditional statements and loops to control the flow of execution within a Java program.

Conditional Statements

Java supports two conditional statements: *if-else* and *switch*.

The If-Else Statement

If the coffee is good, let's drink it, else let's brew another cup. That is exactly what an if-else statement enables you to implement. It verifies whether a logical condition is true or false. If true, the block following the if keyword is executed. If false, the block following the else keyword is executed.

This example compares two integer values:

```
int a = 10;
int b = 5;
int c;

if (a > b)
{
    c = a + b;
}
else {
    c = a - b;
}
/*
The else branch is optional. The following statement is
legal as well:
*/
if (a > b)
{
    c = a + b;
}
```

The alternate form of the if statement can also be used:

```
(a > b)?c = a + b:c = a - b;
```

Information



The curly braces around each single statement in the if-else statement example are not necessary although they can make your code more readable and less error-prone.

An if statement or a loop executes the next statement or block, so the curly braces are needed only if more than one statement is to be executed in context of the if statement or loop.

The Switch Statement

If a condition has more possibilities than true or false, the use of a switch statement is a good alternative. Otherwise you may have to cascade multiple if-else statements, which would be difficult to read and maintain.

This example tests an integer value against several values:

```
public static final int EXCELLENT= 1;
public static final int VERY_GOOD= 2;
public static final int GOOD      = 3;
public static final int BAD       = 4;

int graduation = VERY_GOOD;

switch (graduation){
    case EXCELLENT:
        System.out.println("excellent");
        break;
    case VERY_GOOD:// no break clause, so fall through
    case GOOD:
        System.out.println("good or very good");
        break;
    case BAD:
        System.out.println("bad");
        break;
    default:
        System.out.println("unknown value");
}
```

The variable that is “switched on,” in this case, *graduation*, must be an integer variable (*int*).

The *break* keyword is used to break out of the switch statement. If you do not include a *break* statement, the subsequent statements are performed until the next *break*, which may not be what you want. The statements following the *default* keyword are executed if a *break* statement has not been executed beforehand and is usually used to deal with invalid values.

Loops

Java supports three different loops: *while*, *do while*, and *for*.

The While Loop

The while loop evaluates a logical expression and executes the following block until the logical expression is false:

```
boolean found = false;

while (found == false){
    System.out.println("not found");
    found = true;
}
```

In our example the loop executes exactly once. The first evaluation of the expression is true. The next block is executed. Within the block, found is changed to true. The next evaluation is false. The while block is terminated.

The Do While Loop

The do while loop is similar to the while loop. The only difference is that the block of the do while loop is executed at least once, and the evaluation of the logical expression is at the end of the block:

```
boolean found = true;

do {
    System.out.println("I don't know");
} while (found == false);
```

In this example the block executes once. The block executes under any circumstances because the test of the condition is at the end.

The For Loop

The for loop is usually used if the number of iterations of the loop can be evaluated in advance:

```
for(int i = 1; i < 4; i++) {
    System.out.println("Loop number: " + i);
}
```

Let's look at the definition of the for clause:

int i = 1;	Declares and sets the counter variable to an initial value the very first time, when the loop is entered
i < 4	This expression is evaluated at the start of each iteration of the loop. If the result is true, the block of the loop is executed. If false, the loop is terminated.
i++	The variable i is incremented after each iteration.

In the for clause you can have multiple statements or test multiple conditions, such as:


```
for( int i = 0, j = 10; i < 10 && j < 100; ++i, j *= 2);
```

The Break and Continue Keywords

You have seen how the `break` statement enables you to break out of a `switch` statement. You can also use it to break out of the block in which you are executing. It is usually used to leave a loop. You can also use the labeled *break* statement to break and resume execution at a particular point.

Use the *continue* statement to stop the current iteration of a loop and continue at the beginning of the next iteration.

Now it is time to create some examples in Java. You are about to see how easy it is to use the Scrapbook to test some fragments of code in VisualAge for Java.

The VisualAge for Java Scrapbook

The Scrapbook enables you to evaluate Java expressions. Just type in any expression, highlight it, and execute it. The Scrapbook can contain several pages. You can consider each page of the Scrapbook to be a separate virtual machine that compiles or runs separate code fragments.

Using the Scrapbook

A typical test session using the Scrapbook involves the following steps:

1. From the Workbench menu bar choose **Window**→**Scrapbook** (Figure 12). The Scrapbook window is displayed (Figure 13).
2. Type some Java code (for example, the code shown in Figure 13) into the Scrapbook and highlight the text by using the mouse or the shift and cursor keys.
3. From the menu bar select **Edit**→**Run** or click on the **Run** button on the tool bar.

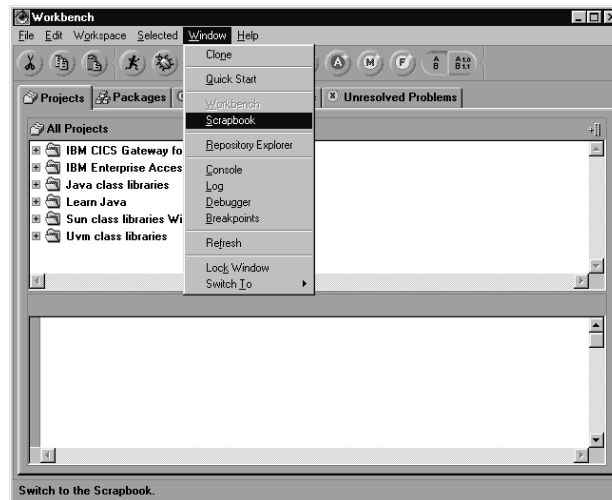


Figure 12. Launching the Scrapbook in VisualAge for Java

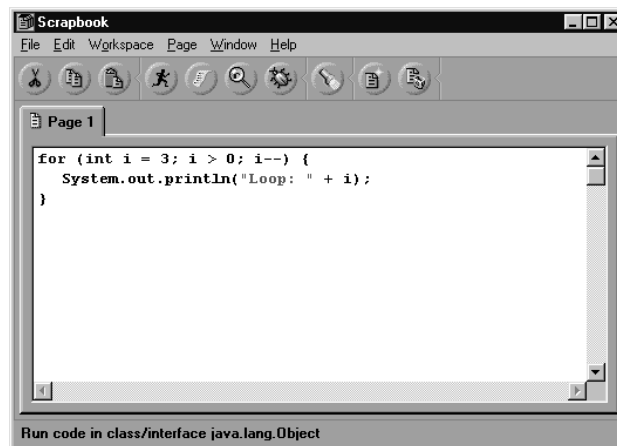


Figure 13. Evaluating Java Code in the Scrapbook

By highlighting the Java statements and choosing **Edit→Run**, you instruct the compiler to compile the statements and execute them immediately. A Console window opens and the output of the `System.out.println("Loop number: " + i);` statement is displayed (Figure 14).

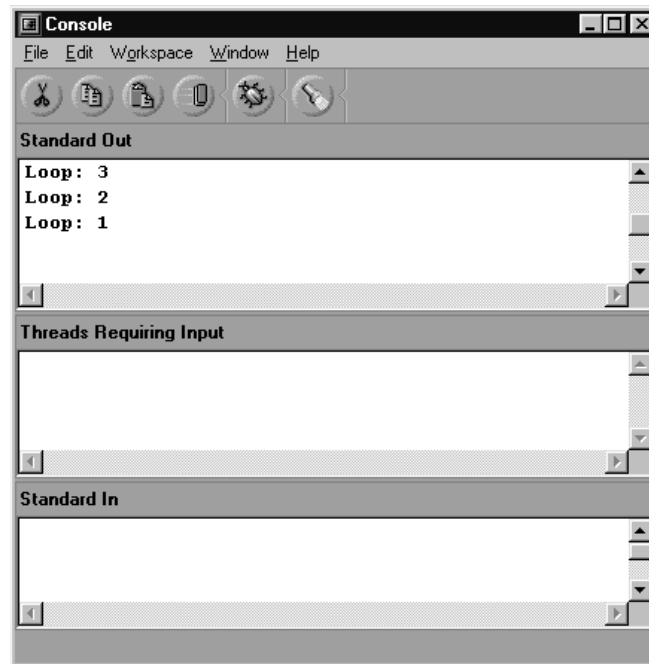


Figure 14. Console Output of the For Loop Executed in the Scrapbook

In these examples you often see a statement similar to this:

```
System.out.println(" any text: " + i);
```

In the previous line of code, the variable `i` is converted to a `String` object, which is concatenated to "any text: ", and `System.out.println()` writes the resulting string to the standard output, which is the VisualAge for Java Console.

Now you can evaluate some of the Java examples in this chapter. Figure 15 shows some examples of using operators in the Scrapbook.

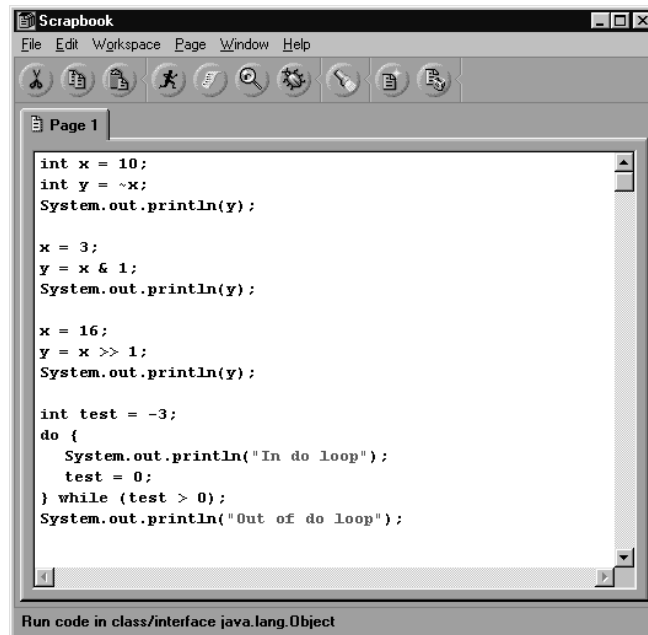


Figure 15. Using Operators in the Scrapbook

Tip



When you execute some code in the Scrapbook, the code runs in the context of the Object class. You can use the **Page→Run In** menu item to change the context in which the code runs.

Correcting Errors in the Scrapbook

If you make a coding mistake, for example, you forget the closing parenthesis at the end of the `System.out.println("Loop number" + i)` statement, VisualAge for Java places a highlighted message where it detects the mistake (Figure 16).

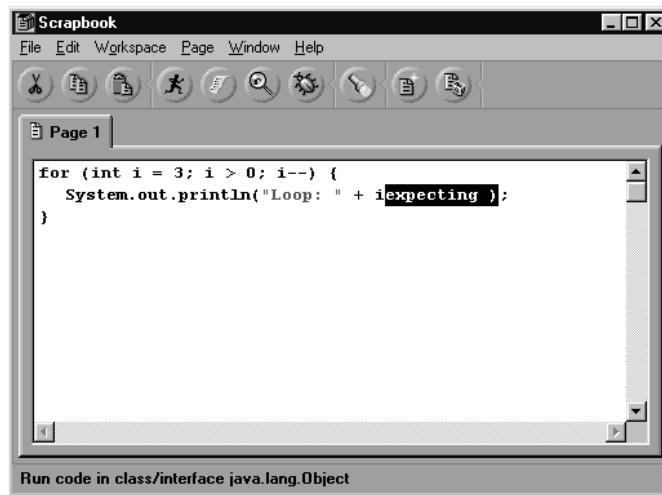


Figure 16. Error Message in the Scrapbook Window

In this case, four simple steps can bail you out:

1. Read the error message to determine what is wrong.
2. Press the Delete key to delete the compiler information.
3. Correct your code.
4. Run your example again.

Tip



Whenever a code fragment is evaluated in a Scrapbook page, that page is made inactive (busy) for the duration of the evaluation. Two visual cues can indicate that a page is busy: An icon displayed in the tab contains a small clock icon, or the status line for the busy page contains the following text: "This page is busy running the selected code."

The page remains busy until the selected code fragment is finished evaluating.

Your page may remain busy because:

- ☐ The code actually takes a long time to evaluate.
- ☐ You are debugging a thread started from that page.
- ☐ You are at a breakpoint reached through the evaluation of code on that page.

To terminate the evaluation of the code and reset the page back to its original state, select **Page→Reset Page...** from the menu bar. Note that all threads, and therefore all windows, started from that page will be stopped and closed.

Summary



Java is an OO language that allows a program to execute on heterogeneous platforms. Although Java is a general-purpose language, its greatest strength is that it extends Web applications with rich functionality.

The basics of the Java language include:

- ☐ Statements: expressions and blocks
- ☐ Variables, keywords and operators
- ☐ Control flow expressions

The Java language supplies predefined classes to manipulate strings and arrays.

The VisualAge for Java Scrapbook facilitates evaluating and testing pieces of code on the fly.

3

Objects and the Java Language

Java is an *object-oriented* language, and, in the next four chapters we explain what that means and why it is important to you. In this chapter you start from the beginning; you learn about *objects* and *classes*, key concepts in object-oriented languages.

To get some practical experience using VisualAge for Java you begin to work on an automated teller machine (ATM) example. Based on the ATM scenario, you implement your first class, the `BankAccount` class, using the VisualAge for Java IDE.

If you are already familiar with object technology or another object-oriented language, in the next chapters you will learn how to implement the concepts you already know in Java. If you are new to objects, be patient. It is a little difficult at first to think in terms of objects rather than the procedural process of a program, but reading the next chapters and following the examples will provide a good basis for understanding objects.

Object-Oriented Programming Concepts

Object-oriented programming facilitates development of complex software systems by breaking them up into a number of much smaller, simpler program elements called *objects*. Historically a lot of hype surrounded this programming technology, and many companies claimed that their product were object-oriented, even though they were not. Such false claims confused consumers, causing widespread misinformation and mistrust of object-oriented technology.

Despite an overuse of the term *object-oriented*, the computer industry is now beginning to realize that, if not a panacea, this technology greatly improves the software development cycle at many points: code quality and maintenance, application scalability, and time to market.

In the sections that follow we introduce you to the first object-oriented concepts you need to understand to read this chapter. In Chapter 4 we describe new concepts that will refine your knowledge of object-oriented programming.

Objects

In our real world, an object is, according to *Webster's Dictionary*:

Something perceptible, especially to the sense of touch or vision.

Indeed, according to this definition, there is a large assortment of objects in our world! Let's take a car as an example. If you ask two people to describe a certain car, they will probably give two completely different answers, on the basis of their knowledge, their way of looking at and evaluating things, and their interests. A passionate driver will tell you about the car's motor and give you many other technical details about the internal workings of the car. A person who has never driven a car will tell you about the color of the car, its estimated length, width, and height, and anything else that is visible. In driving schools, instructors will describe a car by emphasizing its function and explaining how to handle the steering wheel, gearshift, and pedals and how to interpret the indicators on the dashboard. No one, however, can give a "correct" description that encompasses all properties and functions of a car. But, each person will agree that cars share two characteristics; they all have *state* and they all have *behavior*.

Software objects are modeled after real-world objects in that they, too, have state and behavior. A software object maintains its state in *variables* and implements its behavior, using *methods*.

Once it is designed, a software object has a limited number of variables and behaves as defined (but sometimes not as intended) by its methods. Its variables and methods are also called *instance variables* and *instance methods*, to distinguish them from *class variables* and *class methods*, which apply to classes (see Classes on page 46).

Abstraction

Software objects represent real-life objects but are implemented in a manner that is appropriate for a particular problem. A level of *abstraction* decides the model and implementation choices for your objects. Again, let's take a car as an example of an object. For an application that supports car sellers, designers implement the car object with variables that are important for marketing purposes, such as price, horsepower, color, and number of air bags. These attributes are of a rather high abstraction level, because customers usually are not interested in the amount of steel or aluminum that was used to produce the car. For an application that supports the car manufacturing process, however, designers must choose more instance variables for the car object, because engineers are interested in details that are essential to the building of a car. In both applications, the car object adopts only some of its real-world instance variables.

Encapsulation

It is common to represent visually a software object as a donut whose center contains the instance variables and whose periphery displays its methods (Figure 17).

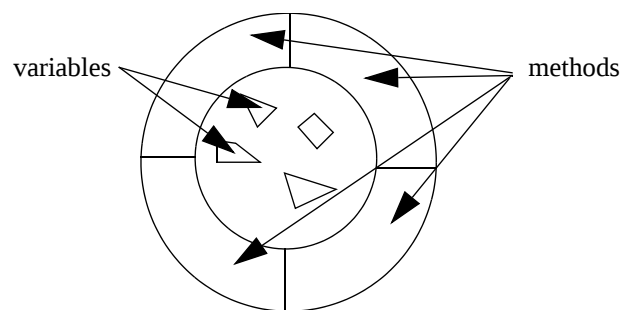


Figure 17. Object Representation

Everything that the software object knows (state) and can do (behavior) is expressed by the variables and methods within that object. For example, a software object that models a real-world car would have variables that indicates the car's current state: its

speed is 80 mph, its engine cadence is 20,000 rpms, and its current gear is the 3rd gear. The software car would also have methods to start the car, brake, accelerate, and change gears (Figure 18).

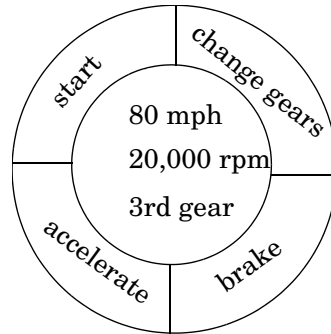


Figure 18. Car Object Representation

Anything that an object does not know or cannot do is excluded from the object. For example, the car object does not have a name, and it cannot jump (usually). Thus there are no variables or methods for those states and behaviors in the car object.

The donut representation of an object makes its variables the center of the donut. The methods surround the variables and act as a protection from the outside world. Methods exposed in the object representation are the only interface available to other objects for accessing the object's variables. We say that the variables are encapsulated in the object to hide its implementation details.

You also find *encapsulation* in real-world object. For example, when you need to stop your car, you press down the brake pedal without knowing the details of the mechanism that make your car stop. The brake mechanism is hidden from you. You only need to use the pedal, which acts as an interface to activate the mechanism.

Similarly in software programs, you do not need to know how an object is implemented; you only need to know which methods to invoke.

Encapsulation provides several benefits regarding the software development cycle:

- ❑ **Modularity:** Because your object encapsulates all of the variables and methods needed to make it work, it is a self-contained entity that can be maintained independently of other objects in your program.

- ❑ **Maintainability:** Because your object hides its implementation details through a well-defined interface, you can decide to change the details without affecting other parts of the program relying on the interface.

Hiding or exposing variables can also be applied to methods of an object and constitutes a design decision. Most object-oriented languages such as Java support access control modifiers that guarantee the level of accessibility of your object members.

Messages

Objects in the real world interact with each other to perform tasks. For example, every morning when you jump in your car to go to your office, you are an object, an instance of a human being class, which interacts with an instance of the car class. By turning the ignition key, you are interacting with your car through its interface, and you activate methods that are defined in its behavior.

Software objects interact in a program in the same way, by sending messages to each other (Figure 19). For example, when a software driver object wants a software car object to execute the start method, it sends a message to the car object.

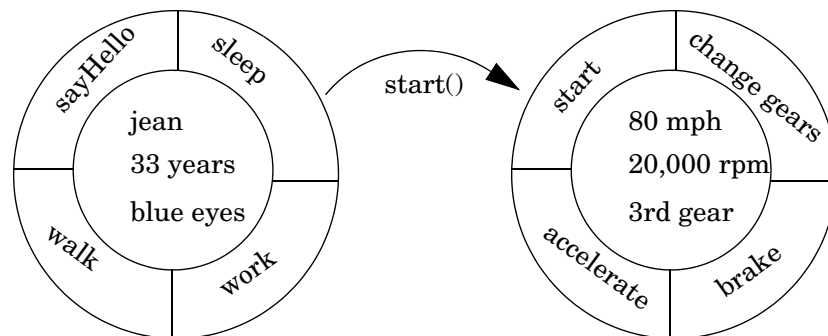


Figure 19. Object Sending Message

Messages consist of three distinctive components:

- ❑ Object receiver to whom the message is sent
- ❑ Method name to be executed for this message
- ❑ Any parameter needed to execute the method

Messages can be sent between objects on the same machine or on different machines.

Classes

If we consult *Webster's Dictionary* again, we find that a class is defined as:

A set or group whose members share at least one attribute.

In effect, in the real world you find many objects that share the same state and the same behavior. For example, cars can accelerate and have a color. Your car is one example of the car class. It shares common state and behavior with other cars but has its own characteristics. For example, its color might be different from the color of your neighbor's car. In object-oriented terminology, your car is an *instance* of the car class.

When car manufacturers create a new model, they define its characteristics in a blueprint. The blueprint is used to produce all cars of the new model. For example, a manufacturer could create a blueprint for a model T car that would define its characteristics, such as its engine type, color, or number of doors. The blueprint would also define the car's behavior to accelerate, change gears, or brake.

In object-oriented terminology, object blueprints are called *classes*. Classes define the common state and behavior shared by different objects or instances of the same class. In car simulator software, you could create a car class that would declare instance variables such as the engine type, color, or number of doors for the car instance. This class would also implement the methods to make your instance accelerate, change gears, or brake. Each instance would have the same instance variables but their value would be different. To simulate a race, you would create from your class as many instances as needed. Then you could invoke instance methods on the different instances created.

Each instance holds a copy of the instance variables declared in the class.

Now that you understand better the primary concepts of the object-oriented technology, let's see how Java relates to them.

What Makes Java Object-Oriented?

In the Java language every variable (except those of the primitive types, such as `int` or `double`) represents a reference or *handle* to an object of a class. You define a Java class by specifying declaring instance variables and class variables and by implementing instance methods and class methods. Any piece of code that you call in a Java program is a method of an object. You cannot define

in Java a method that is not associated with a class. Java supports of course all of the other characteristics that make an “object-oriented” language. Concepts such as *inheritance* and *polymorphism* are introduced in Chapter 6, “Reuse in Java,” on page 103. For now, let’s introduce the ATM example that will serve as a guide for discovering Java with VisualAge for Java.

The Automated Teller Machine (ATM)

Most of us use ATMs regularly. For the remainder of this book you are going to build a prototype for an ATM, sort of. You will not build it in the hardware—you will be smart and build it in the software.

In our scenario, a bank has decided to use dedicated Java machines for its ATMs. The machines will run a Java applet, which is downloaded from a server within the bank. In this way the bank can continually update the ATM interface without the costly expense of upgrading the physical ATM machines.

In this section you work with simplified version of the requirements, analysis, and design of the applet. Then you implement the first class: the BankAccount. In subsequent chapters you continue to build and refine your applet.

Using objects enables you to implement independent components of your final system and refine them throughout the development process. Developers cope with many classes and objects. Some classes and objects directly represent the image of real-life objects; others are metaphors for services that are required to implement the business logic or communicate with either the user of the application or external devices. In large software applications, the associations and interactions among all objects are both difficult to describe and complex, so we must have methods to shed light on that complexity. Without such methods, developers would soon see themselves as “object-disoriented”!

Most of the object-oriented methods divide up the development cycle of an application into three main phases: analysis, design, and implementation. In the next sections we expose the ATM applet system requirements and briefly describe these different phases. Of course, many books and courses are available on object-oriented analysis, design, and application development. If you are going to be working on object-oriented projects, we strongly recommend that you continue your education.

System Requirements

The system requirements describe the functions of your future system. In our case:

- ❑ The ATM machine must allow users to insert their cards, select the appropriate account, and then deposit or withdraw money or check their balance.
- ❑ A bank account must record the ID of the account and the current balance.
- ❑ The checking and savings accounts must support depositing and withdrawing money.

Analysis

The goal of analysis is to understand the problem domain, that is, to clarify *what* the system should provide. So, the first step is to separate the problem domain from the real world. In most cases, you would carry out this first step together with your users and define problem statements that are based on the requirements specification. Then you would arrange these problem statements to form use cases (*Object-Oriented Software Engineering. A Use Case Driven Approach* by I. Jacobson et al.).

Use Cases

One way of recording system requirements is to create *use cases*. A use case is a scenario that the system must support. “Withdrawing money” is an example of use case for the ATM. This use case assumes that your card has already been validated. An *actor* in a use case is any entity (human or machine) which interacts with the system you are analyzing. The use case describes the steps involved in completing the operation. For example, the use case “Withdraw money using a validated card” would involve the following steps:

1. Select one of the accounts listed from the bank.
2. Enter amount of money to withdraw (amount less than the balance).
3. Remove money.
4. Remove card.

Other use cases for the ATM would include:

- ☐ Withdraw money (amount greater than the balance).
- ☐ Deposit money
- ☐ Consult balance
- ☐ Validate card

At the analysis stage, the use cases are rather high level. You do not consider any implementation constraints, except that the system should be affordable and implemented within a reasonable time frame. You simply describe the essentials of the system's functions, regarding the functions as black boxes. Consequently, you take only those objects that directly represent their real-life counterparts; you can find these objects by analyzing the use cases.

Here are some simple rules for finding possible objects, attributes, and methods:

- ☐ To find objects or attributes of objects, look for nouns in your use case text.
- ☐ To find methods, look for verbs in your use case text.
- ☐ Define which items are objects and which are attributes. Attributes tend to be simple values, such as balance or accountId.
- ☐ Assign the found methods to the appropriate objects.

Now you can apply these rules to our Withdraw Money use case.

Nouns - Candidate Classes

The following nouns are potential objects or attributes: bank, account, money, balance, card.

Verbs - Candidate Methods

The following verbs are potential methods: withdraw money, select account, consult balance, and remove card.

Now let's concentrate on the bank account object. It sometimes helps, especially when designing in a team, to describe an object by using its donut representation as shown in Figure 20.

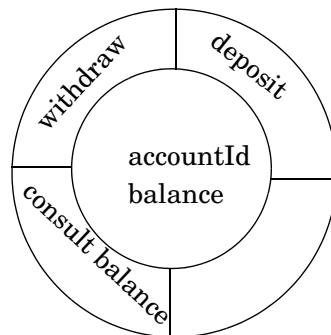


Figure 20. Description of The Bank Account Object

Although `accountId` was not present in the Withdraw Money use case, you can assume it was in the Validate Card use case.

Design

The design stage involves mapping your analysis of the problem domain to the solution domain. The separation between analysis and design can become blurred, but it is good practice to keep them separate.

Some of the components of the design phase are:

- ☐ **System design:** What is the architecture of the solution? What performance will be expected?
- ☐ **Object design:** Completely designing objects for the chosen programming language and environment, including:
 - User interface classes
 - Utility classes (storage classes, system interaction)

In your case you are building a Java applet using VisualAge for Java. For now, let's suppose that the bank accounts are fictitious and created directly in memory but not read from a bank server. Of course, at this stage you should not worry about any other issues like performance, just concentrate on building the ATM applet.

Implementation

The goal of the implementation phase is to translate the design model into the implementation model, that is, the actual application construction. The design and implementation phases are closely coupled as the design prototype gradually evolves toward the final implementation model.

We explain the implementation phase of our BankAccount class in the section Implementing the BankAccount Class on page 60 of this chapter.

The BankAccount Class

Now that you are familiar with objects, classes, and our ATM application, let's create our first Java class.

Creating Java Classes

In Java source code, you define a new class by using the *class* keyword:

```
class BankAccount{}
```

To the new class, BankAccount, let's add the state (instance variables or fields) and the behavior (methods). The simple bank account can be represented as shown in the *class diagram* in Figure 21.

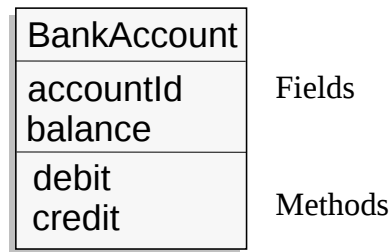


Figure 21. UML Representation of the BankAccount Class

This representation differs from the donut representation we saw in Figure 20. It is inspired by the *Unified Modeling Language* (UML) notation. UML is a fairly recent attempt by some leading figures in the object-oriented design community to produce a unified approach to describing object-oriented systems. For more information about UML, see Appendix B, “UML Notation,” on page 445.

Java classes are defined with the following simplified format:

```
[ modifiers] class ClassName
{
/*
    implementation of class
*/
}
```

Class Modifiers

Modifiers associated with a class can be *public*, *abstract*, or *final*.

The *public* modifier is an access modifier. It defines the classes of objects that can access objects of this class. This modifier declares that the class can be used by any objects. As you will see in Chapter 4, classes are organized in libraries called *packages*. If you do not specify the *public* access modifier, a class can only be used by other classes in the same package in which it is declared.

The two other class modifiers, *abstract* and *final*, are not related to access. They will be introduced in Chapter 6, “Reuse in Java,” on page 103.

Fields

In VisualAge for Java, instance variables are called *fields*. Fields can be a variable of a class, array, or primitive data type. In the bank account you create two attributes to represent the balance and the account identifier. We choose to represent the balance, using the primitive data type *double*:

```
private double balance;
```

We choose to represent the *accountId* using the Java String class:

```
private String accountId;
```

Just like the variables of primitive data types we discuss in Variables on page 21, fields that are of class types can have modifiers such as *protected* and *private*. Fields are usually made *private* to support the encapsulation of the field within the interface of the object. Other objects that want to get or set the value of a field must use specific methods called *getter* and *setter* methods that are exposed in the public interface of the object.

Methods

Methods implement the behavior of our BankAccount object. We initially create two methods to credit and debit the account as well as the getter and setter methods to access and modify the balance field.

Getter and Setter Methods

When you encapsulate the state of an object, the methods that get and set the values of a field are known as accessor and mutator methods or, simply, getters and setters. With getters and setters you can encapsulate logic when accessing to a variable as well as control access to the variable. For example you can check that a bank account will not be overdrawn before you withdraw money from the account.

Although it may seem like extra work, always using getters and setters gives you flexibility in making later changes, complies with the syntax pattern required for defining bean properties (see Introspection and BeanInfo Class on page 249), and helps enforce correct access to fields.

Here are the getter and setter methods of our BankAccount class:

```
public double getBalance()
{
    return balance;
}
private void setBalance(double anAmount)
{
    balance = anAmount;
}
public String getAccountId()
{
    return accountId;
}
public void setAccountId(String anAccountId)
{
    accountId = anAccountId;
}
```

Credit and Debit Methods

The credit method simply adds the anAmount parameter to the balance after checking that the amount is strictly positive:

```
public void credit(double anAmount)
{
    if (anAmount > 0.0){
```

```
        setBalance(getBalance() + amount);
    }
}
```

The debit method checks whether the account has sufficient funds. If the funds are insufficient, a message is printed, and the debit does not take place. The method checks also that you do not try to subtract a negative value—which would actually credit the account!

```
public void debit(double anAmount)
{
    if (anAmount > getBalance()){
        System.out.println("Sorry, there is not enough money");
    }
    else {
        if (anAmount > 0.0) {
            setBalance(getBalance() - anAmount);
        }
    }
}
```

Did you notice that these methods do not have to specify on which account they are acting? In some programming languages you have to specify the `accountId` so that the function or procedure knows which account to update. Not so with methods of an object. Because the methods are associated with one instance, they act only on the balance associated with that object.

Methods are defined with the following simplified format:

```
[modifiers] returnType methodName([parameter list])
```

Method Modifiers

The modifiers associated with a method include access modifiers and other modifiers. The other modifiers are explained in Chapter 6, “Reuse in Java,” on page 103.

Access modifiers define which objects can invoke a method as follows:

- ❑ **Public:** The method can be invoked by any object.
- ❑ **Protected:** The method can be invoked within the current object, by objects within the same package, and by subclasses of this object’s class.
- ❑ **Private:** The method can be invoked only by objects of the same class.

When no access modifier is specified for a method, the package access is used by default. In that case, the method can be invoked by any object in the same package.

Notice that the `setBalance` method above is defined as `private`. This is a further example of encapsulation. It means that other objects must use the bank account's public interface of `credit` and `debit` to change the balance of the account. The balance cannot be set directly.

Return Type

A method must declare a return type, which can be any primitive type or class in Java. The return type can also be `void`, which indicates that the method does not return a value.

An example is the `getBalance()` method, which returns a `double`.

Method Name

The name of a method must follow the same rules as Variable Names on page 23. You learn more about special names for methods and overriding method names and signatures in Chapter 6, “Reuse in Java,” on page 103. A method *signature* is the combination of its name and parameter list.

Parameter List

A method declaration includes a parameter list, even if the method takes no parameters. In that case the list is simply empty. Otherwise, the list is a comma-separated list of the names and types of the parameters. For example, both the `credit` and the `debit` methods have one parameter, `amount`, of type `double`.

In Java all methods must be associated with a class; you cannot create them in isolation as you would create functions or procedures in other programming languages.

Figure 22 shows the complete code for the `BankAccount` class.

```
public class BankAccount{
    private double balance;
    private String accountId;

    public void credit(double anAmount) {
        if (anAmount > 0.0) {
            setBalance(getBalance() + anAmount);
        }
        return;
    }
}
```

```
    }  
    public void debit(double anAmount)  
    {  
        if (amount > getBalance()){  
            System.out.println("Sorry, there is not enough money!");  
        }  
        else{  
            if (anAmount > 0.0) {  
                setBalance(getBalance() - anAmount);  
            }  
        }  
    }  
    public double getBalance() {  
        return balance;  
    }  
    private void setBalance(double anAmount) {  
        balance = anAmount;  
    }  
    public void setAccountId(String anAccountId) {  
        accountId = anAccountId;  
    }  
    public String getAccountId() {  
        return accountId;  
    }  
}
```

Figure 22. The Complete BankAccount Class

Wrapper Classes

Java also provides wrapper classes that enable you to use primitive types, such as integers, in places where you need an object. Java provides a corresponding wrapper class for each of the primitive types. For example, the *Integer* class is used to create Integer objects. You can use wrapper classes with classes such as the *Vector* class which allows you to store objects in dynamic vectors. To store integer values you would first need to wrap them with the Integer object:

```
int i = 10;  
Vector v = new Vector();  
v.addElement(new Integer(i));
```

The BankAccount Objects

BankAccount objects are instances of our BankAccount class and are the elements in the ATM application that do the work. Classes are the elements you work with at development time to define

your objects. When your program is running, the objects interact to provide the function. The class simply defines what an object knows (its state) and its behavior (methods).

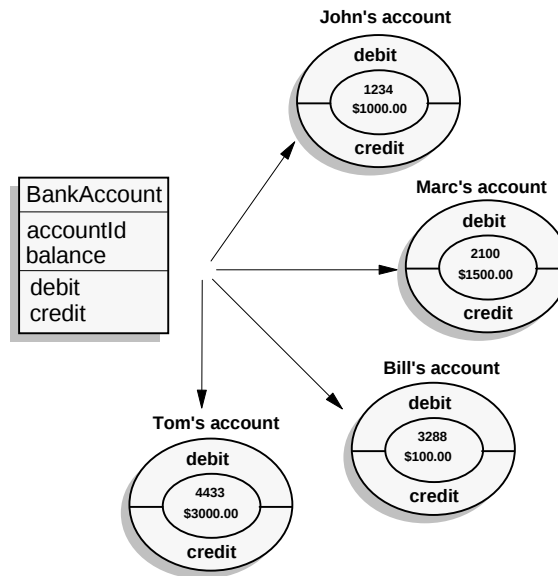


Figure 23. The BankAccount Class and Some BankAccount Objects

Figure 23 depicts BankAccount objects instantiated at run time from the BankAccount class. Each object holds different values for its instance variables, but all objects share the same method code, which is defined in the BankAccount class.

Static Fields and Static Methods

In Object-Oriented Programming Concepts on page 42 you learned about instance variables, instance methods, class variables, and class methods. The fields and methods of our BankAccount class are instance variables and instance methods; that is, they are associated with an instance: an object.

Java lets you use the *static* qualifier to create class variables or class fields and class methods. Class fields and class methods belong to and act on the class as a whole. They are shared among all instances of the class. For example, a bank might set an interest rate that is the same for all accounts. This rate could be created as a static variable and would be shared by all accounts:

```
static float simpleInterestRate = 0.08;
```

Now any bank account object can access the `simpleInterestRate` variable.

Static methods also relate to the class rather than the instance. These methods can only access static variables, and they can be invoked without first creating an instance of a class.

Using Objects within Your Programs

Now that you have designed your `BankAccount` class, you need to learn how to instantiate `BankAccount` objects from it and use these objects in your programs.

Object References

In a Java program you declare variables or *object references* of a particular class. Then you assign them to an instantiated object or another variable. For example, in the following code:

```
BankAccount account;
```

`account` is an object reference: its only purpose is to refer to an object. For now it is just a place holder for objects of the `BankAccount` class but it is not an object. Next we create a new `BankAccount` object and assign it to our object reference:

```
account = new BankAccount();
```

For now, think of the `new` operator as simply the way of instantiating an object from a class by returning a new object of the class that follows it, in this case, `BankAccount`. You will learn all about the `new` operator in Chapter 5, “Lifecycle of Java Objects”.

You could also declare another account variable:

```
BankAccount same_account;
```

and then assign it to the previous variable:

```
same_account = account;
```

In the above line of code, you work with the same account object, you just have two references to it.

Message Sending

Once you can refer to the new `BankAccount` object, using `account` or `same_account`, you can send it messages by calling its methods. To access the fields and methods of the object, use the following syntax:

```
account.debit( 100.00); // debit the account $100
```

or

```
int balanceVar = account.getBalance(); // assign the account balance
// to an integer variable.
```

The syntax for referring to the fields and methods of an object is the name of the variable followed by a dot (.) followed by the name of the field or the method. At the end of this chapter and in the rest of the book, we show you many examples of using objects within your programs.

Within the body of a class, you can simply reference fields and methods directly: `debit(100)` or `balance`.

Object Assignment

When you assign an instance of an object to an object reference, you are not making a copy of the object. Consider the following code:

```
BankAccount b1, b2;
b1 = new BankAccount();
b2 = b1;
b2.credit( 2000);
```

After this code is executed, both `b1` and `b2` have a balance of 2000; there is only one object but two object references.

Well, it is now time to implement our `BankAccount` class, using `VisualAge for Java`. In the next section we show you how to use the `SmartGuide` features to do just that, with a few mouse clicks!

Implementing the BankAccount Class

To create the BankAccount class with VisualAge for Java, follow these steps:

1. Create the new class.
2. Create the new data fields.
3. Create getters and setters.
4. Create debit and credit methods.
5. Test the class in the Scrapbook.

During each step, you are guided by a specific SmartGuide that produces the code automatically for you.

Creating a New Class

You create the BankAccount class in the *Learn Java* project and the *ch03* package:

1. Launch VisualAge for Java.
2. If you already had VisualAge for Java started, you can open the Quick Start window from the Workbench menu by selecting **Window**→**Quick Start**.
3. In the VisualAge for Java Quick Start window, check the **Create a new class/interface** radio button and click the **OK** button.
4. The **SmartGuide - Create Class or Interface** dialog opens, and you enter the information for your new class (Figure 24):
 - Project: *Learn Java*
 - Package: *ch03*
 - Class Name: *BankAccount*

Select the **Browse Class when finished** checkbox and then click **Finish**. The Superclass field in the dialog box is the name of the parent class of the class you create. All Java objects exist in a hierarchy starting from the Object class. This will become clear after reading Chapter 6, “Reuse in Java”.

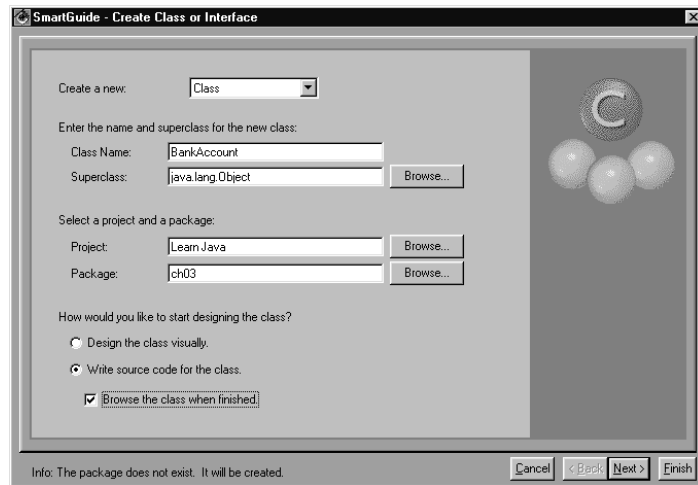


Figure 24. Defining the Project, Package, and Class Name

5. VisualAge for Java prompts you as to whether to create the package. Click **Yes** to create the class and open the Class Browser as shown in Figure 25. If you forget to select the **Browse Class when finished** checkbox, only the Workbench is displayed. In this case, select the new class: **BankAccount**, and click on the **Selected**→**Open** menu item.

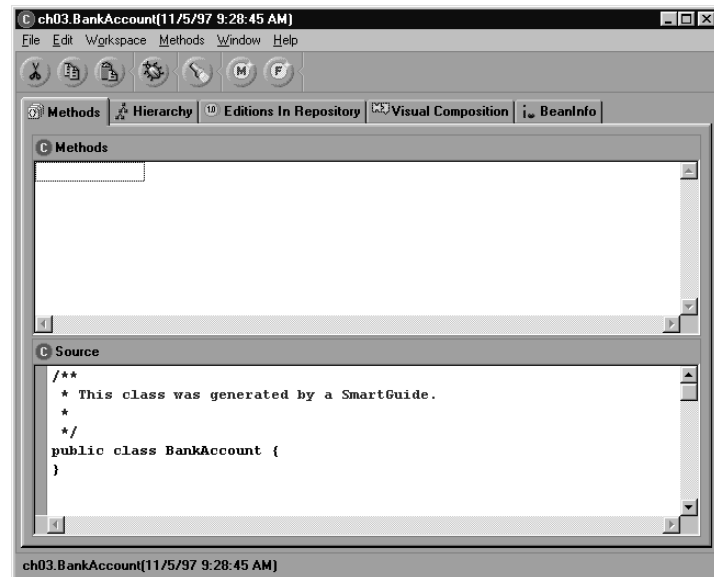


Figure 25. BankAccount Class in the Class Browser Window

Read This

In the rest of this book you will use the following naming conventions for the exercises:

- ☐ Project: **Learn Java**
- ☐ Package: **ch##**, (ch03 for the examples in this chapter).

6. In addition, when writing Java code in the Source pane of the Workbench, make sure you save it by pressing the Ctrl+S keys or by using the **Save** option from the pop-up menu of the Source pane or by selecting **Edit**→**Save** from the Workbench menu bar.

Creating the Balance Field

Now that you have defined the project, package, and class, you can add the fields:

1. From the Class Browser's menu bar choose **Methods**→**New Field** or select **Create Field** from the tool bar.
2. In the SmartGuide - Create Field dialog (Figure 26), define the Field Name as *balance*. Select *double* as the Field Type, click the **private** radio button in the Access Modifiers group, then click **Finish**.

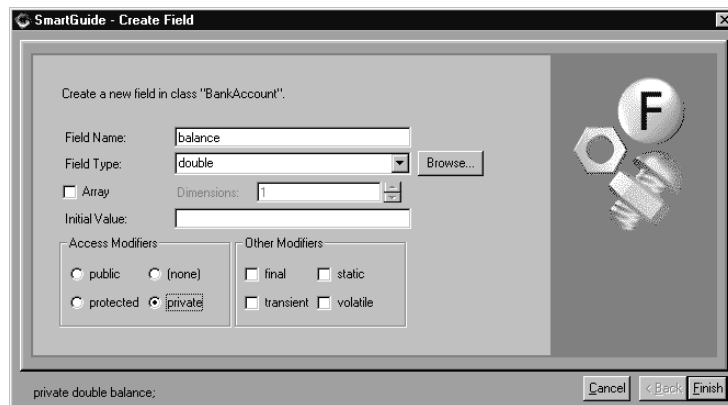


Figure 26. Creating a New Field

Creating the accountId Field

To create the `accountId` field of type `String`, repeat the steps in Creating the Balance Field on page 62. Make the setter for `accountId` *public*.

Now you have two instance fields for your new class and you can see the Java code generated for the fields in the class browser.

Creating Getter and Setter Methods for the Balance Field

Let's now create the methods to access and modify the balance of a `BankAccount` object.

Creating the *getBalance* Method

1. From the Class Browser's menu bar, choose **Methods**→**New Method** or use the **Create Method or Constructor** tool bar button.
2. In the Smart Guide - Create Method define the Method Name as *double getBalance()*. Make sure that the **public** radio button in the Access Modifiers section is selected and click **Finish**.

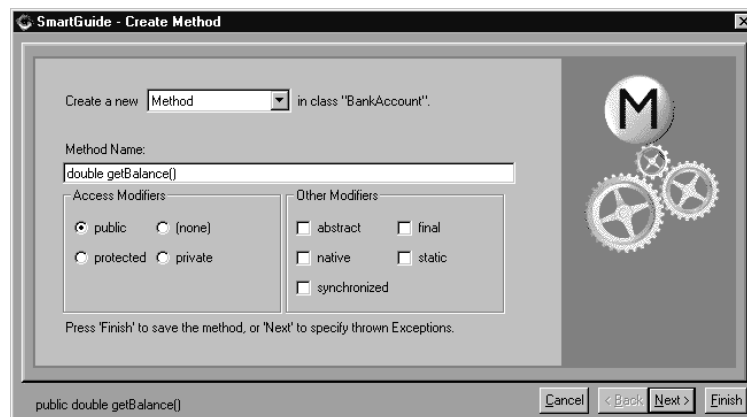


Figure 27. Creating the Getter Method

3. In the Source pane of the Class Browser change the body of the `getBalance` method to the code shown in Figure 28. Now your getter returns the balance field.
4. From the window menu bar choose **Edit**→**Save**.

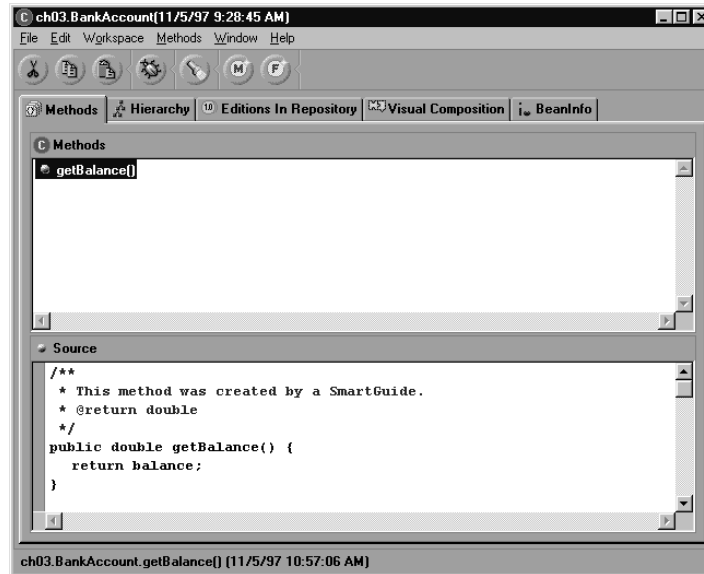


Figure 28. Code of Getter Method

Creating the setBalance Method

1. From the Class Browser's menu bar, select **Methods**→**New Method**.
2. In the SmartGuide - Create Method window (Figure 29) define the Method Name as *void setBalance(double anAmount)*. Make sure that the **private** radio button in the Access Modifiers section is selected and click **Finish**.

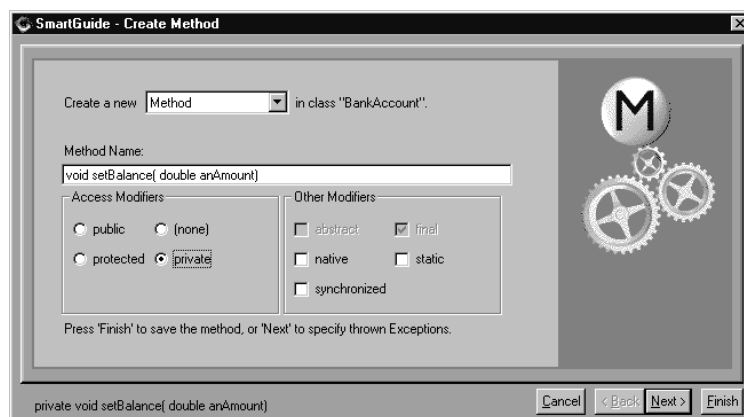


Figure 29. Creating the setBalance method

3. In the Source pane of the Class Browser change the code as shown in Figure 30. Your setter sets the balance attribute to the given value, specified in the parameter amount.
4. From the window menu bar select **Edit**→**Save**.

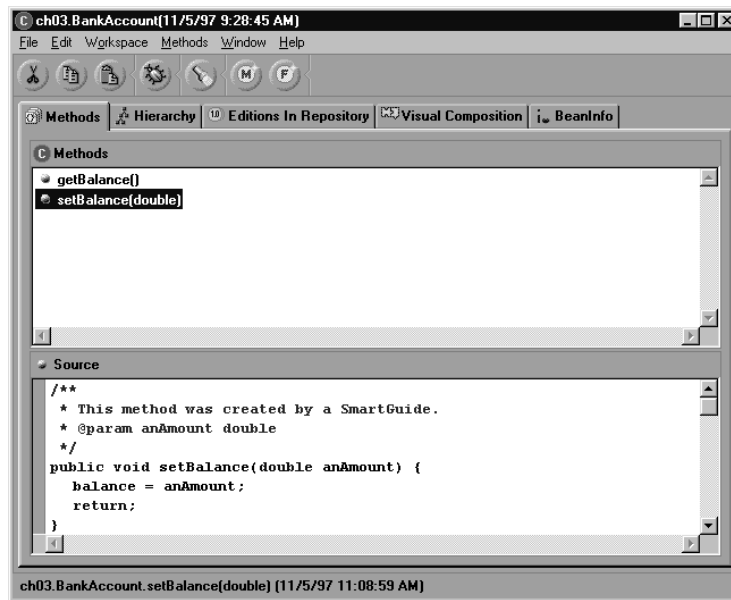


Figure 30. Code of the Setter Method

Creating the Getter and Setter Methods for the accountId Field

Repeat the steps in Creating Getter and Setter Methods for the Balance Field on page 63, for the accountId field. Make the setter for accountId public.

Correcting Mistakes

When you enter incorrect code in VisualAge for Java, one of two things can happen:

- ❑ When you save your code, the parser spots a syntax error, for example, a missing parenthesis. In this case you must find and fix the error before you save your code.
- ❑ When saving your code, the incremental compiler finds a semantic error. In this case you can save the class declaration or method. If the code is within a method, the method will be flagged with a red X, and the method's class will be flagged with a grey X. If the incorrect code is within a class declaration, the class will be flagged with a red X. You can find all

incorrect code in the workspace by opening the *Unresolved Problems* Page using the tab of the same name in the Workbench. Figure 31 shows an example of a coding error in the `setBalance` method.

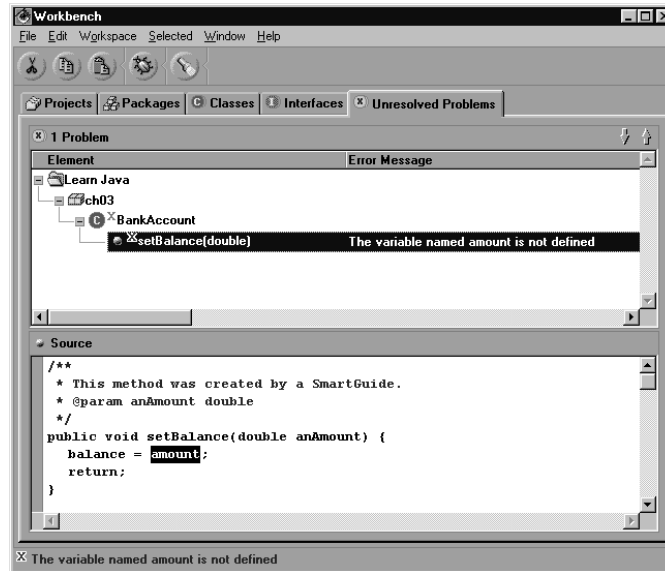


Figure 31. Unresolved Problems Page: Incorrect Code

Creating Debit and Credit Methods

Now let's create the credit and the debit methods for the `BankAccount` class. The steps are almost identical to those for creating the getters and setters for the field.

Tip



There are several ways to create new methods. You have already created new methods using the SmartGuide - Create Method. You can also create new methods by typing over existing methods. Be careful though; the default VisualAge for Java IDE behavior is to replace method you are typing over. By selecting the **Saving Replaces Methods** checkbox in the Behavior page of the Workspace→Options dialog (Figure 32), you can over methods to create new methods. Make sure you deselect the **Saving Replaces Methods** checkbox if you do want to change the method names!

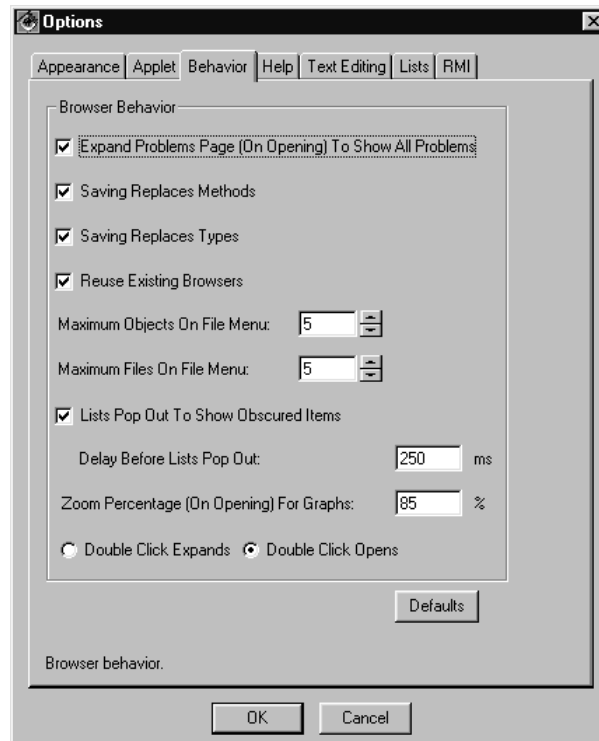


Figure 32. VisualAge for Java IDE Behavior Page

Creating the Credit Method

1. Create a new method in the BankAccount class; for Method Name enter *void credit(double anAmount)* and make sure the **public** radio button in the Access Modifier group is selected. Click the **Finish** button and then in the Source pane of the Workbench correct the code as shown in Figure 33.
2. From the window menu bar select **Edit**→**Save**.

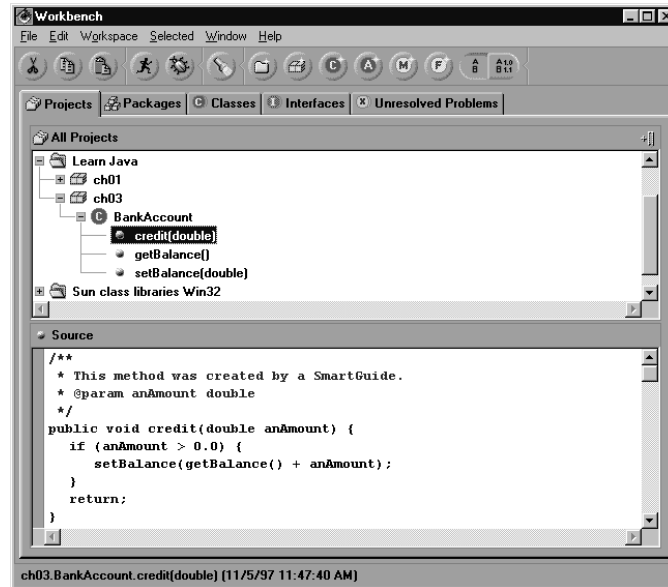


Figure 33. Code of Credit Method

Creating the Debit Method

1. Create another new method but for the Method Name enter *void debit(double anAmount)* and make sure the **public** radio button in the Access Modifier group is selected. Click the **Finish** button and then in the Source pane of the Workbench window correct the code as shown in Figure 34.
2. From the window menu bar select **Edit**→**Save**.

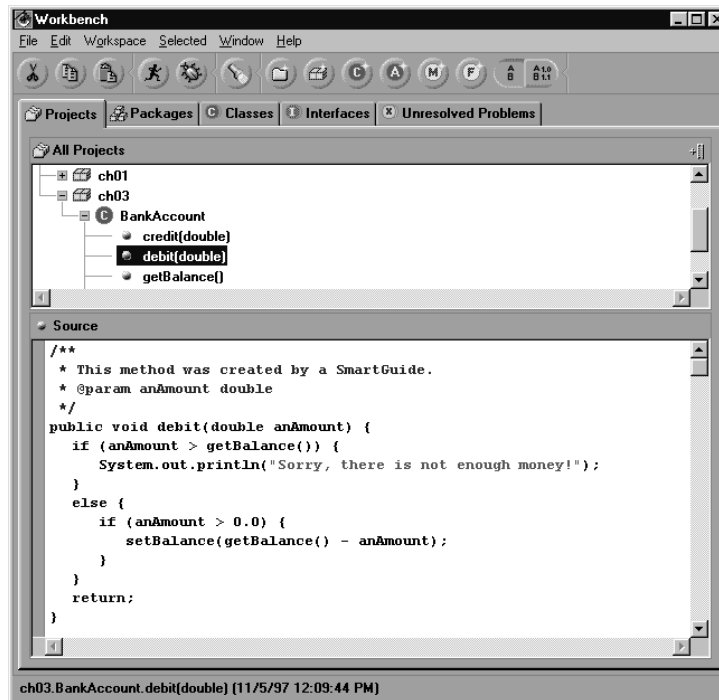


Figure 34. Code of Debit Method

Testing the Class in the Scrapbook

Congratulations! You have created the first class of the ATM application, using VisualAge for Java. Now let's test the class in the Scrapbook by defining and instantiating a BankAccount variable and testing all the methods you have written so far.

1. Open a Scrapbook window and type the following code:

```

ch03.BankAccount account;
account = new ch03.BankAccount();
System.out.println("current balance is: " +
    account.getBalance());
account.credit(100.0);
System.out.println("current balance is: " +
    account.getBalance());
account.debit(40.0);
System.out.println("current balance is: " +
    account.getBalance());
account.debit(300.0);
  
```

Notice the `ch03.BankAccount`? The `ch03` is the name of the package in which you defined the `BankAccount` class. You will learn all about packages in Chapter 4.

2. Select the code in the Scrapbook (control-A selects all the code) and run it (control-E is the shortcut for run).
3. The output is shown in the Console window (Figure 35).

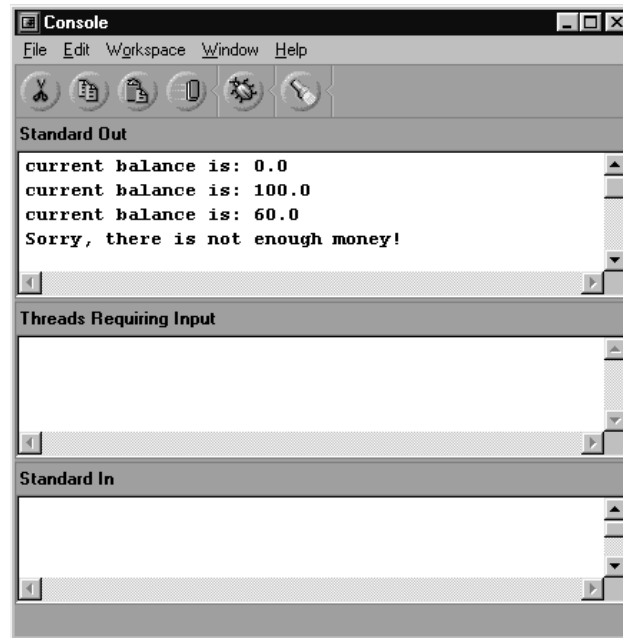


Figure 35. Console Output of the Test Program

Summary



In this chapter you learned about classes and objects, the building blocks of object-oriented programming.

You also learned about object references and how to use them within your Java programs.

An object shares the behavior of all objects of the same class, but each object can have a different state. VisualAge for Java facilitates object-oriented development with Java by providing SmartGuides that guide you through the development process of programming a Java class.

4

Organizing Your Code

Packages are Java's way of organizing classes and interfaces that are intended to work together. VisualAge for Java supports the creation of packages and enforces good programming practice by making you explicitly define packages.

In this chapter you learn what Java packages are and how you can use them. You also become familiar with the project concept, which, in the VisualAge for Java environment complements packages.

What Is a Package?

You created your first package when you created your first class in VisualAge for Java. The SmartGuide assisted you in creating a project and at least one package within that project. You have also created a few classes, all belonging to packages. Thus you probably have some idea of what packages are.

Packages are like libraries in other programming languages. They provide a way of grouping logically related classes or interfaces and avoiding naming conflicts. (Interfaces have not been introduced yet, but for now assume that an interface is a collection of method definitions.)

Avoiding naming conflicts is very important especially when you download Java classes from the Web and want to ensure that they do not conflict with your own classes.

Classes and interfaces in a package are usually logically related. It is the programmer's responsibility to decide which packages need to be created and which classes or interfaces should be included in each package. It is also possible to have subpackages: packages named within packages. The end result is that the names of Java packages form a tree structure.

More about Access Modifiers

In Chapter 3 you learned about access modifiers for classes and class members. Packages also relate to access modifiers. When methods, fields, or classes are defined without an access modifier, they are accessible only within their package. When methods or fields of a class are defined as protected, only classes within the package or subclasses of that class can access them. At last, when methods or fields of a class are defined as private, only the class itself can access them.

Table 4 summarizes the different access possibilities.

Table 4. Access Possibilities

Access Modifier	Class	Subclass	Package	Other
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

The Class column lists the options for a class to access its own members. The Subclass column lists the options for a subclass to access the members of its base class. The Package column lists the options for a class within a package to access members of another class of the same package. The Other column lists the options for any class to access members of a specific class.

Packages in File-Based Java

The JDK uses directories to organize packages on your file system. When you compile a .java file which contains class definitions, the Java compiler produces one .class file for each defined class. If the classes are defined within a specific package, you must create a directory of the same name as the package name. For example, if you define a package called `account`, and you have a class called *BankAccount* defined in this package, it is your responsibility to create a directory called `account` and to put the source file (`BankAccount.java`) into this directory. When compiling your class, the Java compiler produces the compiled `BankAccount.class` file into the same directory. If the `account` package has subpackages, subdirectories for each of the packages have to be created under the `account` directory.

CLASSPATH Environment Variable

Once your .class files are located in their proper directories, you can run your application or applet by invoking the Java interpreter. The Java interpreter locates the classes needed to run your code by using the CLASSPATH environment variable. CLASSPATH contains a list of directories that tells the Java interpreter where you have installed various classes and interfaces.

When you run your code from the Workbench, VisualAge for Java sets the CLASSPATH environment variable to ensure that all classes and interfaces required to run your code are accessible by the Java interpreter.

Creating Packages

In Java you declare a class to be part of a package by using the *package* keyword at the top of your Java source file. Here is an example:

```
package account;
public class BankAccount{
    private string accountId;
    private double balance;
}
```

If you create several classes in only one source file, they all belong to the package that you define at the top of the file. However, only one of the classes in this case is declared *public*, and the file must be named after that class. Therefore you must have at least one file for every public class.

The definition of a subpackage contains both the name of the package and the subpackage to which the class belongs:

```
package account.utility;  
  
public class MyFileStorage{  
    ...  
}
```

This example defines a subpackage called *utility* for the *account* package. To access this class, the Java interpreter assumes that the following path (on a Windows or OS/2 platform) is defined on your file system:

account\utility\MyFileStorage.class

The directory in which the account directory resides must be referenced in the CLASSPATH environment variable (on a UNIX, Windows, or OS/2 platform). If you do not specify any package name in your source file, Java uses the default package: it expects the current working directory to contain the classes.

Using Packages

With VisualAge for Java or the JDK environment, you have two ways of using a class from another package: you can refer to the class by using the fully qualified name (the default in code generated by VisualAge for Java) or use the *import* keyword.

For example, to create a new package, *calculators*, and a class, *LoanCalculator*, that uses the *BankAccount* class from the *account* package and the *MyFileStorage* class from the *utility* package, you would use the fully qualified names:

```
package calculators;  
  
public class LoanCalculator extends java.lang.Object{  
    private account.BankAccount acc;  
    public static void main (String args[]){  
        acc = new account.BankAccount();  
        account.utility.MyFileStorage storage =  
            new account.utility.MyFileStorage();  
        ...  
    }  
}
```

Writing the fully qualified name of each class can be tedious, especially when writing programs that use many different classes. To make writing programs easier, you can tell the Java compiler that

you want to use classes of selected packages in your program, thus eliminating the need to use fully qualified names. Use the *import* keyword as follows:

```
package calculators;
import account.*;    // import all classes from the
                    // account package
import account.utility.MyFileStorage; // import the
                                    // MyFileStorage class
                                    // from the utility
                                    // package

public class LoanCalculator extends java.lang.Object{
    private BankAccount acc;
    public static void main (String args[]){
        acc = new BankAccount();
        MyFileStorage storage = new MyFileStorage();
        // ...
    }
}
```

As this example shows, you can either import all classes of a package or be more selective. Note that importing a package does not import the subpackages of that package; you have to import them separately or, more typically, use the start notation.

Packages in VisualAge for Java

VisualAge for Java does not use files or directories. Instead, the packages and classes are kept in the *workspace*. The workspace contains all program elements with which you are currently working. Rather than creating directory structures, VisualAge for Java organizes its packages and classes internally. If you export classes, the directory structure is created on your file system. This approach makes managing packages and classes simple. Although classes of your projects are kept in the workspace, resources of those projects are kept separately in your local file system in the directory:

```
\ibmvjava\ie\project_resources\project_name.
```

Another reason for using a workspace-based approach is version control. VisualAge for Java comes with a version control system that is integrated in the IDE. Using the workspace enables you to load and unload different editions of program elements. Version control is discussed in Chapter 9, “Managing Your Code”.

Creating Packages

To create a package with VisualAge for Java, you select a project that will contain the new package. Then, select **Selected→Add Package** from the Workbench menu bar. A SmartGuide window is displayed (Figure 36) and you can type the name of the package to add in the Create a new package named: entry field. Then click the **Finish** button to create the package inside your selected project.

Whenever you create a class using a SmartGuide, a package name is required. If you create a class without specifying a package, the default package is used. Using the default package is not recommended. If you use the default package, you cannot access package members from other packages. If you specify a new package name, that package will be created for you.

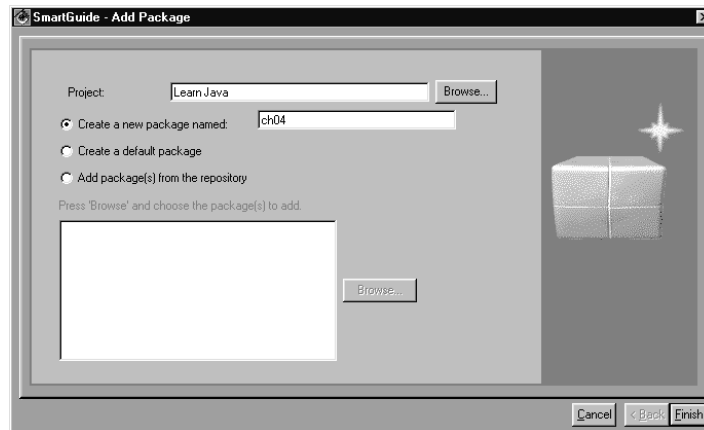


Figure 36. Creating a ch04 Package in the Learn Java Project

Once you create new classes from a selected package, the package name is not shown in the code. In the VisualAge for Java environment the package can be visualized only through the Workbench or the browser.

Read This

Java package names, by convention, consist of lower-case letters, except for the leading com prefix.

Try to use short but descriptive names (a difficult task, to be sure). When you have several levels of subpackages, all with long names, you soon end up with a name like:

```
mypackages.thisisforprivateuse.testingenvironment
```

If you are distributing packages publicly, the recommended format is your reverse Internet domain followed by your package hierarchy. Because your domain is unique on the Internet, it uniquely identifies your classes. For example, packages in IBM VisualAge for Java classes are of the form:

```
com.ibm.ivj...
```

Using Packages in Your VisualAge for Java Code

Fully qualified class names are used in the code generated by VisualAge for Java. For your own code, it is usually much easier to use the import facility.

Tip

The java.lang package is actually imported by default, so you do not have to import it explicitly to be able to refer to the classes of the package such as the String class.

If you already know which packages you will need within your class when you are creating it, the Attributes dialog (Figure 37), which opens when you click **Next** in the Create Class or Interface SmartGuide, enables you to specify the packages to be imported.

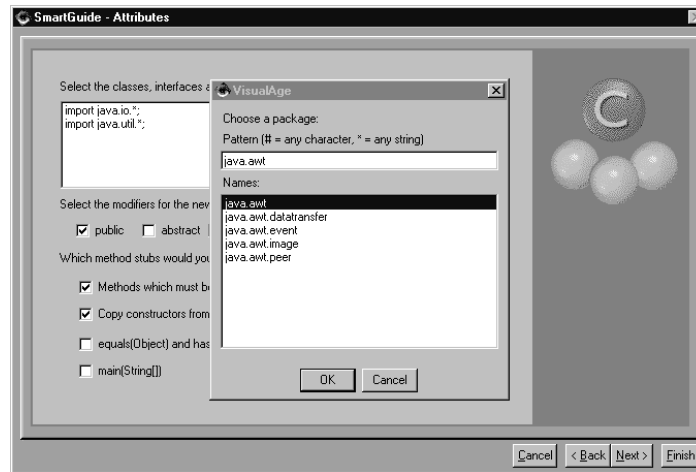


Figure 37. Importing Packages with the SmartGuide

Use **Add Class** and **Add Package** buttons to browse and select classes and packages to import into your class.

You can also add the import statement manually by editing the class declaration.

Importing and Exporting Java Code

You can import and export Java classes and resources to and from the VisualAge for Java environment. In the sections that follow we detail how you can import legacy Java code or export the code you develop with VisualAge for Java.

Importing Java Code to VisualAge for Java

Importing code in the VisualAge for Java environment is not the same as using the *import* keyword in order to avoid coding fully qualified class names in your Java programs. You can import and export different file types.

Java Source Code

You can import Java source code from the file system. The code is compiled as it is imported, and package hierarchies are retained.

You should export Java source code when you want to share it with developers using a different development environment or if you want to edit the source code of your classes using an editor different than the one provided with VisualAge for Java.

Java Class Files

Java class files are exported when you are ready for testing or production. These files are interpreted by the Java VM. Class files can also be imported into VisualAge for Java. However, the imported class cannot be modified, and the source code, as you would expect, is not visible.

Java Archive Files

Java Archive (JAR) is a platform-independent file format that enables you to compress a Java applet and its resources (.class files, images, sounds, and so on) into a single file. JAR files are a good way of distributing Java applets, applications, and their supporting resources. You can install them on a Web server where Web browsers can access them. You can export a complete VisualAge for Java project as a JAR file.

JAR files can also be imported into VisualAge for Java. The classes are compiled and placed in a project. Any supporting resource files are placed in a resource directory under:

```
\ibmvjava\ide\project_resources.
```

VisualAge for Java Interchange Files

Interchange files are used to exchange Java classes built with VisualAge for Java between different VisualAge for Java environments.

When you import an interchange file, it is loaded into your repository. You then have to load it into your workspace from the repository.

Any program elements that you export by using an interchange file must be versioned first. Versions and the repository are fully explained in Chapter 9, “Managing Your Code,” on page 181.

Using the Import SmartGuide

To import Java files in VisualAge for Java, select **File→Import** from the menu bar of the Workbench window. A SmartGuide prompts you for the Project name and the type of files you are importing (Figure 38).

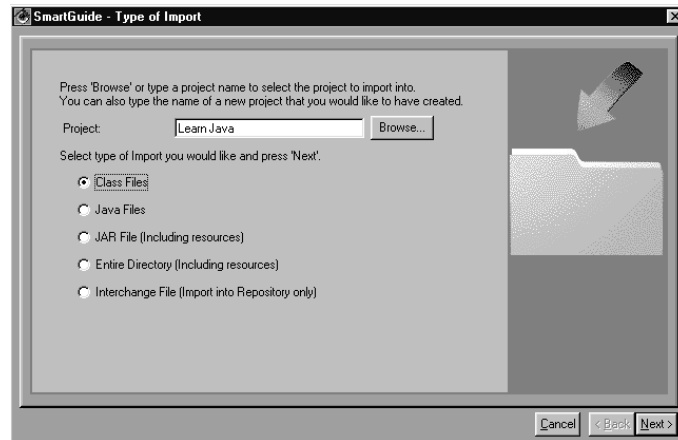


Figure 38. SmartGuide - Type of Import

Once you have selected the type of file to import, select the project that will include the imported packages and their classes. Then browse and select the files and directories to import (Figure 39).

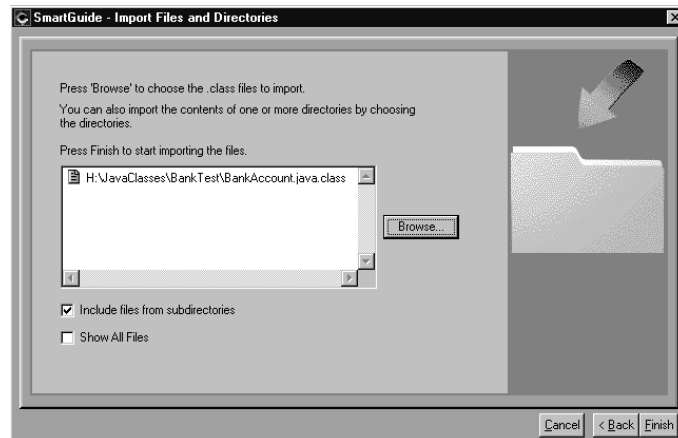


Figure 39. Importing Classes

Exporting Java Code from VisualAge for Java

After creating an applet or an application, you may want to export its class files. You will want to export classes to:

- ☐ Install them on a Web server
- ☐ Install them as a Java application
- ☐ Share them with others outside the VisualAge for Java environment.

You can select classes to be exported in several ways. In the Workbench window, you can select one or more projects, packages, or classes. When you select a package, all classes in the package are exported. When you select a project, all classes of all the packages within that project are exported. You can also export classes individually. After selection, select **File**→**Export** from the Workbench menu bar. The SmartGuide - Type of Export dialog (Figure 40) opens to assist you in specifying the type of files you want to export.

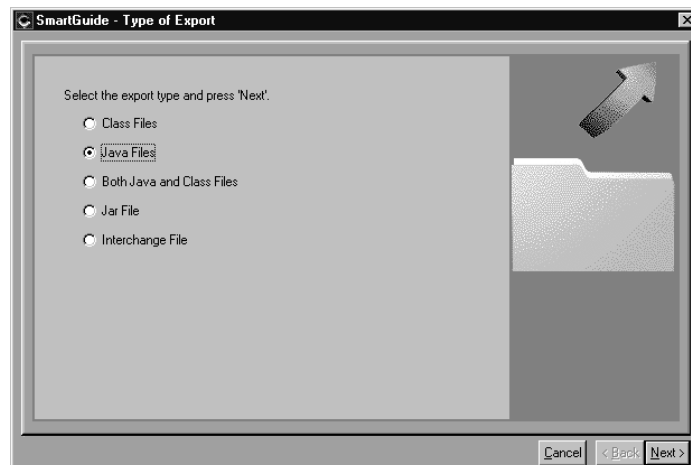


Figure 40. SmartGuide - Type of Export

Once you specify the type of files to export, click **Next>** and enter the directory to which to export the files (Figure 40). If you are exporting Java class or source files, you can choose between putting all the files in the same directory or in directories that correspond to their package names; the latter is the recommended method. VisualAge for Java creates the directory structures for the packages automatically and the package statement in the files that are exported.

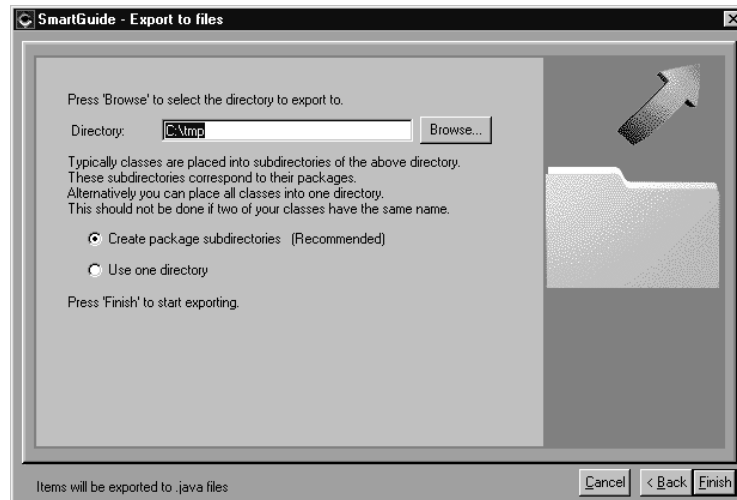


Figure 41. Exporting Classes

Projects in VisualAge for Java

As we discuss in Chapter 1, “Introduction to the Environment”, projects provide a way of organizing your Java code at the highest level. Whenever you create a new package, you must place it in a new or existing project. Projects do not have any equivalent in Java. VisualAge for Java provides them so that you can organize your packages.

The first time you start VisualAge for Java and go to the Workbench, you see several projects in your workspace, all of which contain several packages. Table 5 lists several projects and their contents. Some of the projects listed may be in the repository rather than the workspace.

Table 5. VisualAge for Java Projects and Their Contents

Project	Contents
Java class libraries	java.* packages, which contain the Java 1.1.1 class libraries
Uvm class libraries	uvm.* packages, which enable and support the VisualAge for Java VM and environment
Sun class libraries	sun.* packages, which are extensions to standard java.* packages

Table 5. VisualAge for Java Projects and Their Contents

Project	Contents
Sun JDK examples	Sample applets and classes from Sun
Sun BDK examples	Sample JavaBeans from Sun
IBM Java examples	COM.ibm.ivj.* packages containing sample code created with VisualAge for Java

Java Class Libraries

The Java Class Libraries Project contains the Java 1.1 packages. Programs written with these packages are portable to any other Java 1.1 environment without shipping any other additional packages. For more information about the Java class libraries see the library documentation. The packages include:

- ❑ **java.lang**
This package is automatically imported into Java classes. It includes, for example, the String and Object classes, the wrapper classes for primitive data, and basic system support classes (System, Compiler).
- ❑ **java.io**
The java.io package contains classes that assist in input/output operations, such as reading and writing files (classes such as File, FileDescriptor, FileReader). Different types of streams are included (such as DataOutputStream, ByteArrayOutputStream, and PrintStream).
- ❑ **java.text**
This package contains classes that enable locale-sensitive parsing and formatting of data types such as Date and Decimal. The formatting classes include NumberFormat, DecimalFormat, and DateFormat.
- ❑ **java.math**
This package contains the BigInteger and BigDecimal classes.
- ❑ **java.beans**
Includes classes that support JavaBeans (JavaBeans are introduced in Chapter 11).
- ❑ **java.net**
Includes classes that provide basic networking support, such as socket and URL handling. Classes in this package include InetAddress, DatagramSocket, ServerSocket, and URL.
- ❑ **java.awt**
This package includes classes that provide the graphical user interface in the Java abstract windowing toolkit (AWT). The

user interface classes include: Panel, Frame, Font, MenuBar, TextField, Button, Dialog and Canvas. Event support classes are in the java.awt.event package. Support for image handling is provided in the java.awt.image subpackage.

❑ **java.applet**

The Applet class in this package provides the basic behavior for applets. The Applet is the class you inherit from when writing your own applets.

❑ **java.security**

This package provides security functions for Java classes.

❑ **java.rmi**

This package contains the RMI support. It includes classes and interfaces that support distributed Java application development.

❑ **java.util**

This package contains general-purpose utility classes such as Date, Calendar, StringTokenizer, and Vector.

❑ **java.sql**

This package contains support classes and interfaces for accessing relational databases using JDBC. Classes include Connection, DriverManager, and Statement.

Summary



Java uses packages to organize related classes and interfaces. In file-based Java environments, packages are organized in a directory structure. In VisualAge for Java, packages are organized internally. Only when exported does the directory structure become visible.

VisualAge for Java introduces the concept of project to organize your packages. Several projects are provided initially with the product. They contain packages such as the Java 1.1 packages, environment packages, and samples of using both standard Java and VisualAge for Java.

5

Lifecycle of Java Objects

In this chapter you continue your education and learn how objects are instantiated, how to define your own methods to initialize your objects and how objects are discarded within your Java programs. You also learn how to use VisualAge for Java to implement a better version of the `BankAccount` class.

Instantiating Objects

As we explain in Chapter 3, "Objects and the Java Language", an object is instantiated from a class. To instantiate an object from the `BankAccount` class, you use the *new* keyword:

```
BankAccount account = new BankAccount();
```

In the above statement you declare a variable, or object reference, `account`, of type `BankAccount`. By using the *new* operator, you create a new object that is an instance of the `BankAccount` class, and by using the assignment operator you assign the `account` vari-

able to the newly created instance. You can also declare the variable and then create the object and assign the reference to it using separate statements:

```
BankAccount account;  
account = new BankAccount();
```

The first line declares the variable, and the second line assigns to the variable a value, an instance of the `BankAccount` class.

Initializing Fields

The `BankAccount` has two fields: `accountId` and `balance`. What happens to them when a new object is created? Initialization values of fields depend on their type. Remember that in classes such as `String`, the initial value of an object reference is `null`, and the variables of the primitive data types are initialized to the default value of each type (for example, for `double`, the initial value is `0.0`).

It is good programming practice to initialize field values to a state that is expected by a user of that object. In the bank account example, a minimum amount of \$20 is required to start an account. You could make this `initialAmount` a static variable in the class declaration:

```
public class BankAccount {  
    private static double initialAmount = 20.0;  
    private double balance;  
    private String accountId;  
}
```

To initialize the account's fields to initial values, you could define an `init()` method that would initialize the `balance` to the `initialAmount` and the `accountId` to a value which you would supply to the method as a parameter:

```
public void init(String anAccountId){  
    accountId = anAccountId;  
    balance = initialAmount;  
}
```

Now whenever you want to use a `BankAccount` object, you could call the `init` method in your program after creating the object:

```
BankAccount account = new BankAccount();  
account.init("1234-5678");
```

There is a problem with this approach. What happens if a developer that is using your `BankAccount` class does not know that the object has to be initialized with the method you wrote for that purpose? You definitely need another way of initializing objects right from the start.

Constructors

A *constructor* is a method that constructs objects of its class. Because it is a method, you can define a constructor in the same way that you would define any other method. Well, almost! There are a few differences. A constructor must have the same name as the class itself, and it does not explicitly define a return value: it implicitly returns a new object of the class. Otherwise, constructors can do the same things as other methods, such as instantiate other objects, call other methods, and set field values. You can define for the `BankAccount` class a constructor that initializes the object's fields to initial values for the `BankAccount` class:

```
BankAccount(String anAccountId){  
    accountId = anAccountId;  
    balance = initialAmount;  
}
```

Once you have defined this constructor, you create instances of the `BankAccount` class, using the *new* keyword and supplying an `accountId`:

```
BankAccount account = new BankAccount( "1234-5678");
```

This time, the constructor you defined above is invoked, the `balance` and `accountId` are initialized in the constructor, and there is no need for the `init()` method we discussed previously.

In fact, you have been using constructors from the very beginning. You just were not aware of it. Java provides a default constructor that is invoked when a class does not define its own constructor. The default constructor does not take any arguments; it creates an instance of your class and initializes the field values to their default initial values.

Static Fields Initialization

The purpose of a constructor is to initialize fields of an object when the object is created. Because static fields are shared among all objects of the same class, initializing these fields in a constructor would result in resetting their value for all objects already cre-

ated, each time a new object is created. For this reason, you initialize static fields directly in the class definition, as you did for the `initialAmount` field of `BankAccount`.

When the static field is a composite field such as an array, you can use a *static initializer* to do the initialization for you in the class definition. Consider the following example of the `BankAccount` class, which contains an array of interest rate objects (for this example, we assume that the you have previously created an `InterestRate` class):

```
public class BankAccount {
    private static double initialAmount = 20.0;
    private double balance;
    private String accountId;
    static final int nInterestRate = 5;
    static InterestRate[] interestRateList =
        new InterestRate[nInterestRate];

    static {
        for (int i = 0; i < nInterestRate; i++) {
            interestRateList[i] = new InterestRate((i + 1) * 0.005);
        }
    }
}
```

In this example, the static initializer creates an array of five interest rate objects ranging from 0.5 to 3%. The number of interest rate is initialized in the final static field, `nInterestRate`. Then, the static initializer creates the interest rate objects by calling the `InterestRate` constructor in a loop. Notice that the *final* modifier makes `nInterestRate` a constant.

Information



If you define a constructor with no arguments, the default constructor is *overridden*. Overriding occurs when you define a method with the same signature as a method in one of the parent classes. See Chapter 6, "Reuse in Java".

If you provide a constructor that takes arguments, the default constructor can no longer be used. Users of your class might try to use the default constructor, so it is good programming practice to provide a constructor with no arguments as well as the constructors that require arguments.

Overloading a Constructor

You are free to define as many constructors as you like as long as the arguments that you pass are different. For example, you can define a constructor that initializes the balance to a different value as well as setting the accountId field:

```
BankAccount(String anAccountId, double amount)
{
    accountId = new String(anAccountId);
    balance = amount;
}
```

You use this constructor by passing both arguments:

```
BankAccount account = new BankAccount("1234-5678", 1000.00);
```

Remember that a method's signature is the combination of its name and the parameter list. Creating in the same class more than one method with a different signature (with the same name but different arguments) is called *overloading* a method. In this case you overload the BankAccount constructor by creating the two different versions above.

Static initializers are not real methods and do not have arguments, so they cannot be overloaded.

Calling One Constructor from Another

Once you have overloaded a class constructor, you can significantly reduce your code by calling one constructor from another.

Consider the following example of your BankAccount class where you define two constructors:

```
public class BankAccount {
    private static double initialAmount = 20.0;
    private double balance;
    private String accountId;

    BankAccount() {
        this("000-000");
    }

    BankAccount(String anAccountId) {
        accountId = anAccountId;
        balance = initialAmount;
    }
}
```

The default constructor reuses the code defined in the second constructor to create a bank account with a default identifier of “000-000”. The call to the alternate constructor must be the first statement in the default constructor. The *this* keyword is used to call one constructor from another defined in the same class.

The *this* Keyword

When you create two objects (a and b) from two different classes that implement the same *f()* method, you may wonder how Java calls the correct method from each object. The “trick” lies in a secret first argument, which is passed to the method when it is invoked. This secret argument is a reference to the object from which you call the method. The argument can be accessed within the object’s method by using the *this* keyword. Thus, the *this* keyword represents the reference of the object for which the method is called.

The *this* keyword is used to invoke alternate constructors from another constructor. More generally, you can invoke from an object’s method any method that requires the calling object by passing the *this* keyword as a parameter.

Copy Constructor and References

Let’s imagine for a moment that you create a fictitious Bank class that would hold only one BankAccount object:

```
public class Bank {  
    private BankAccount account;  
  
    BankAccount(BankAccount anAccount) {  
        account = anAccount;  
    }  
  
    String toString() {  
        return “A Bank with ... “ + account;  
    }  
}
```

Let’s suppose that you refine your BankAccount class by providing a *toString()* method:

```
public class BankAccount {  
    private static double initialAmount = 20.0;  
    private double balance;  
    private String accountId;
```



```
BankAccount() {  
    this("000-000");  
}  
  
BankAccount(String anAccountId) {  
    accountId = anAccountId;  
    balance = initialAmount;  
}  
  
String toString() {  
    return "A BankAccount " + accountId;  
}  
}
```

Then you create a main method to your Bank class, which would create a Bank object and its BankAccount object:

```
public static void main(String args[]) {  
    BankAccount aBankAccount = new BankAccount("123-456");  
    Bank aBank = new Bank(aBankAccount);  
    System.out.println(aBank);  
    aBankAccount.setAccountId("345-678");  
    System.out.println(aBank);  
    return;  
}
```

If you execute this code, you get the following result in the Console window:

```
A Bank with ... A BankAccount: 123-456
```

```
A Bank with ... A BankAccount: 345-678
```

By changing the account identifier of aBankAccount, you change the account identifier of the bank account held by aBank.

As we mention at the beginning of this chapter, when you declare a variable to hold an object created by the new operator, the variable becomes a reference to this object. In this regard, aBankAccount is a reference to the BankAccount object that you create in the first line of the main method. When you create a Bank object in the second line of the main method, you use the constructor defined in Bank. This constructor uses the aBankAccount reference to initialize its account field. At this moment account and aBankAccount reference the same object. Thus, when you change the account identifier of aBankAccount in the fourth line of the main method, you also change it for the account field of the aBank

object. Furthermore, even though the main method has no access to the `account` field through `aBank`, it has direct access to the object through the retained reference, `aBankAccount`.

Such a violation of the autonomy of the `Bank` class is acceptable if you ensure that the `aBankAccount` will not be changed, but most of the time it introduces design issues in your code.

To prevent such issues, you can use a copy constructor for your `BankAccount`. A copy constructor initializes a `BankAccount` object from the source object supplied as argument:

```
BankAccount(BankAccount anAccount) {  
    accountId = anAccount.accountID;  
    balance = anAccount.balance;  
}
```

Notice that the signature of a copy constructor is defined as `C(C)`, where `C` is a class. You can modify the constructor of your `Bank` class by using the `BankAccount` copy constructor:

```
public class Bank {  
    private BankAccount account;  
  
    Bank(BankAccount anAccount) {  
        account = new BankAccount(anAccount);  
    }  
  
    String toString() {  
        return "A Bank with ... " + account;  
    }  
}
```

By calling the copy constructor of `BankAccount` in the `Bank` constructor, you explicitly ask Java to allocate a new `BankAccount` object from the heap and copy the object referenced by `anAccount` into it. Thus, if you modify the object referenced by `anAccount`, your `Bank` object will not change.

Notice that in the `BankAccount` constructors you have been simply assigning the passed reference to your `accountId` variable, using:

```
accountId = anID;
```

This is fine when the variable is a `String` because strings are immutable—you cannot change them. If this object were of any other class, you would want to consider making a copy of the object in your constructor so that the object would not be changed without your knowledge.

Passing Arguments in Methods

By passing an outside object to a constructor, you pass a reference to the object. Thus, any modifications you make to the object inside the constructor apply directly to the outside object. The same applies to any method defined in a class.

Programming languages can be divided into two categories according to whether they pass method arguments by reference or by value.

Passing Arguments by Value

Programming languages that pass arguments by value make a copy of the argument in the stack allocated by the program before entering the method. When the program exits the method, the original argument is retrieved from the stack. Thus, any modifications performed on an outside object passed as an argument inside of the method have no effect on the outside object when the method terminates.

Passing Arguments by Reference

Programming languages that pass arguments by reference pass a memory address of the arguments to the method. Thus, any modification made to an outside object passed as an argument inside the method is applied directly to the outside object and remains when the method terminates.

How about Java?

In Java, all arguments are passed by value. So you would expect that any object you pass to a method cannot be modified when the method terminates. That makes sense, but do not forget that Java manipulates object references only. Thus when passing an object as a parameter, you pass a reference to the object. Of course, the reference is copied into the stack and is thus passed by value. But once your method has access to the object reference, any change made to the argument is made directly on the outside object.

Object Cloning

The C++ language passes everything by value and in that regard resembles the Java language. Under several conditions, C++ calls the copy constructor of objects that you pass as method arguments and prevents those objects from being modified when the method terminates. Although the copy constructor can be very useful in many occasions, it is never called automatically by Java. Thus, to prevent outside objects that you pass as method arguments from being modified, you can clone them before performing any modifications. All modifications are performed on the object clone, and your original object can be restored when the method terminates. For more information regarding the cloning process, refer to the JDK documentation.

Access Modifiers and Constructors

The default modifier for a constructor is *package*; that is, this constructor can be called by any other class of the same package. You can also use *private* and *protected* modifiers on constructors. Defining a constructor as *private* indicates that you can only use it as a private method, so it can be used only by the class itself. A *protected* constructor can be used only by the class itself, by its subclasses, or by classes in the same package. No modifier indicates that the default modifier applies.

When the Work Is Done: Finalizers

Whereas constructors give life to objects, finalizers serve the opposite purpose: you use them when the life of an object ends, for example, to free up common resources.

Java objects require memory just like the elements of any other programs. When new objects are instantiated, the memory is allocated from the *heap*, which is a block of memory that Java controls. When objects are no longer referenced, the memory chunks allocated to them are returned to the heap to be used by new objects.

What does “no longer referenced” mean? An object is referenced when at least one variable references it. When the last variable which references of an object A goes out of scope or when another object is assigned to the last variable, the object A can be finalized and its resources freed up.

Unlike C++, Java does not have a concept of *destructor*, a method that is called automatically when an object is destroyed. The reason lies probably in the fact that in Java objects are automatically destroyed when they are no longer referenced. A process, called *garbage collector*, runs in the background to reclaim the memory as necessary from the objects that are no longer referenced. However, although this process manages the memory for you, you may still need to perform extra steps, such as closing a file or a socket opened by your object, before your object gets destroyed. This is why the `finalize()` method is always called before your object is “garbage collected.”

To use the `finalize()` method, you just have to include it in the definition of your class. Here is an example:

```
public class BankAccount{
//  other code removed for clarity

    // a finalizer method that prints a message
    protected void finalize(){
        System.out.println("Bye for now!");
    }
}
```

After the `finalize()` method has been run, the object is garbage collected. The memory allocated to the object is returned to the system for use by other objects.

You do not usually control when the garbage collector runs, so do not rely too heavily on the `finalize()` method running at a particular time. It is also possible to explicitly invoke finalizers of any unused objects, using `Runtime.getRuntime().runFinalization()`.

You can also invoke the garbage collector directly, using `System.gc()`.

Creating Constructors and Finalizers with VisualAge for Java

Now it is time to put VisualAge for Java to work. The Smart-Guides can assist you in creating both constructors and finalizers for your classes.

Before starting, let's organize our code by creating a new package, `ch05`, which contains the code already written for the `BankAccount`. We also have to add the `initialAmount` field to our class.

Copying Packages

To create ch05, you copy the ch03 package to a new package named ch05 in your Learn Java project:

1. Select the ch03 package.
2. Choose **Copy** from the context menu (displayed when you click the right mouse button) or the **Selected** menu bar item.
3. In the Copying Selected Items dialog (Figure 42), ensure that Learn Java is entered in the Copy To field and click **Finish**.
4. Enter ch05 in the entry field in the Copy dialog and click **OK**.

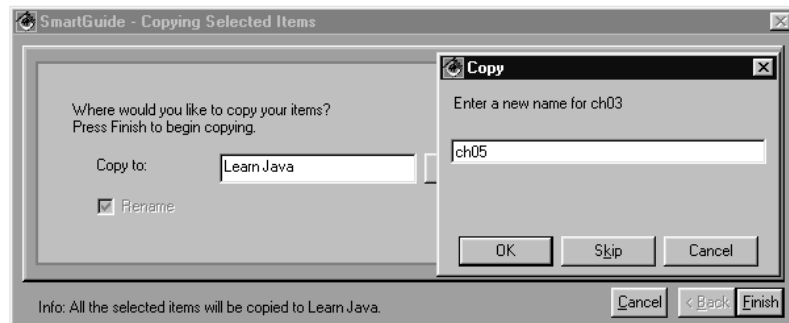


Figure 42. Copying the BankAccount Class

Adding the initialAmount Field

In ch05.BankAccount, create an initialAmount field, define it as private static double, and choose 20.0 as its initial value (see Source pane in Figure 43).

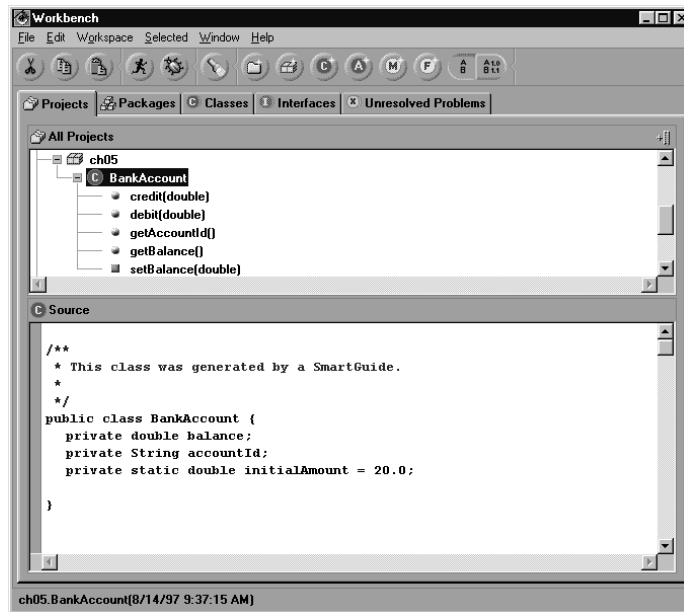


Figure 43. The BankAccount Class in Package ch05

Creating Constructors with VisualAge for Java

To implement a constructor with one argument of type String:

1. Select the BankAccount class, in package ch05, in the Workbench (Figure 43).
2. Select **Selected**→**New Method** from the menu bar to open the SmartGuide (Figure 44).
3. Select **Constructor** in the Create a new drop-down list box, and change the **Constructor Name** field to: *BankAccount(String anAccountId)*.

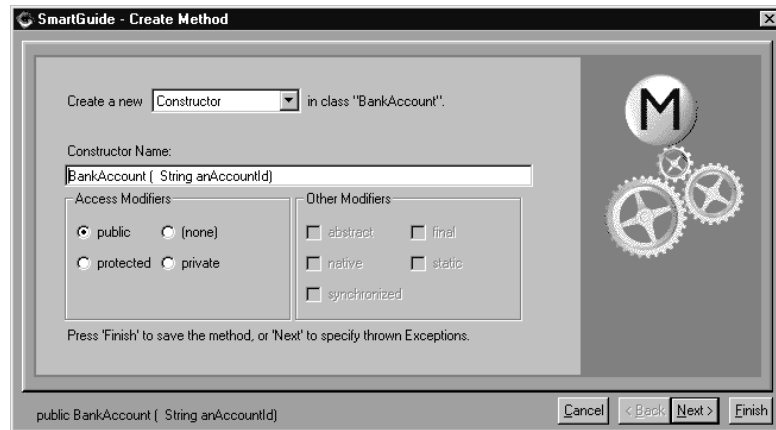


Figure 44. Defining Constructors in VisualAge for Java

You do not have to change the access modifiers because this constructor is public, available to all users of this class. Click **Finish** and look at the constructor's code in the Source pane of the Workbench window.

The default constructor does not do much, so add the assignment of the `accountId` field and the `balance` field as shown in the following code:

```
public BankAccount (String anAccountId) {
    accountId = anAccountId;
    balance = initialAmount;
}
```

Select **Edit**→**Save** from the menu bar or use the control-S key shortcut to save and compile the modified constructor.

To test your constructor, select **Window**→**Scrapbook** from the menu bar to open the Scrapbook. Type in the following statements:

```
ch05.BankAccount account = new ch05.BankAccount("1234-5678");
System.out.println(account.getAccountId());
```

Select the above statements and select **Edit**→**Run** from the menu bar. You should see the Console window (Figure 45) showing the `accountId` of the `BankAccount` object.

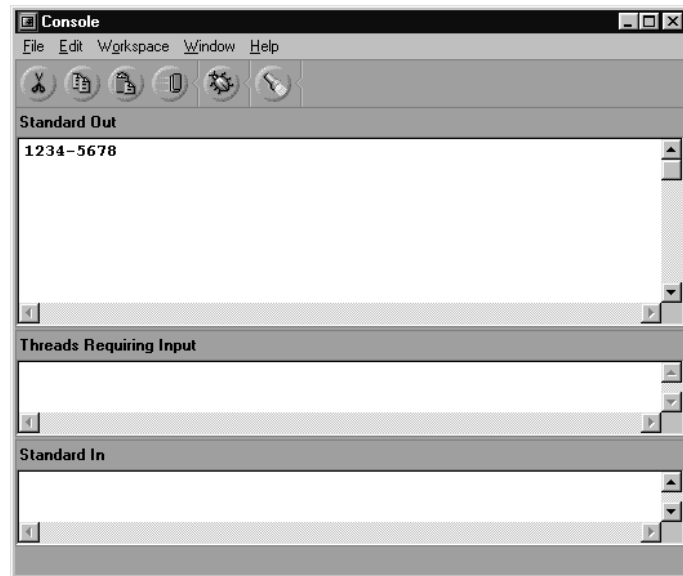


Figure 45. Testing Your Constructor

Copying Constructors

You duplicate the constructors of a superclass in a class you define only when you define the class by accessing the second panel of the SmartGuide and selecting the **Copy constructors from superclass (Recommended)** checkbox (Figure 46). Beware that this option has nothing to do with the copy constructors that we explained in “Copy Constructor and References” on page 90.

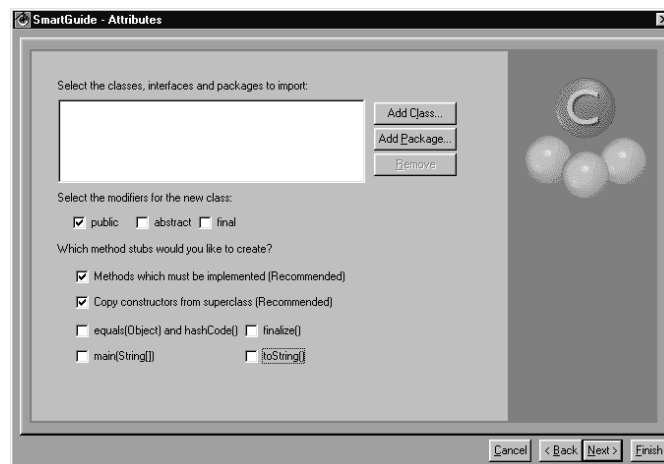


Figure 46. Creating a Copy Constructor or a Finalizer

Notice that, using the same page, you can create `finalize()` and a `toString()` methods. These methods are also defined from their base class methods.

Defining Finalizers with VisualAge for Java

You can define a `finalize()` method with VisualAge for Java in two ways:

1. Use the **finalize()** checkbox in the Attributes dialog of the Create Class SmartGuide.
2. Add the `finalize()` method the way you would add any other method, using the Method Properties SmartGuide or overwriting an existing method.

If you use the first method, you must use the Create Class or Interface dialog. Clicking the **Next>** button takes you to the second page (Figure 46) of the SmartGuide. From this page you can check on the creation of the `finalize()` method stub.

In this exercise, because the class is already created, you create the `finalize()` method the same way you create any other method:

1. In the Workbench, select the `ch05.BankAccount` class, and then click on **Selected→New Method** from the menu bar. The SmartGuide opens to assist you in creating a new method.
2. In the Method Name text field of the SmartGuide, type *protected void finalize()*.
3. Click **Finish**.
4. Modify the method that you see in the Source pane to look like this:

```
protected void finalize() {  
    System.out.println("Bye for now!");  
    return;  
}
```

Save the method, and you are ready to test your finalizer.

If you still have the Scrapbook window open, you can use the same two lines of code you used to test your constructor. After the code is run, the Console window displays the same results as it did for the constructor, but this time, there is an additional line of output.

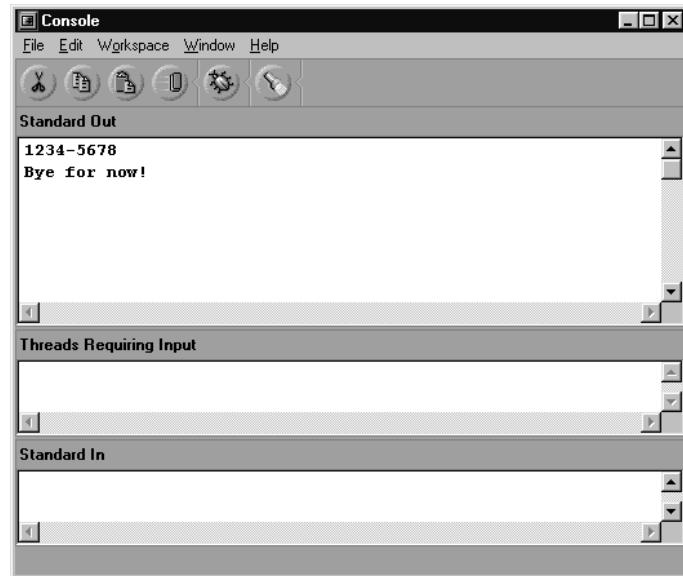


Figure 47. The Finalizer at Work

Tip



If you do not see the text that you defined in the `finalize()` method displayed in the Console window, your system is probably busy doing something else. The `finalize()` method will be invoked eventually.

Summary



The Java language provides a way of defining how objects are created by means of a *constructor*, a special method that has the same name as the class that defines it. A constructor can take any number of arguments, which can include information about the object should be instantiated. You can define as many constructors as you like, as long as the arguments are different. The default constructor takes no arguments, but as soon as you define a constructor of your own, you lose the default constructor and you must re-create one if needed.

The Java VM executes a method called *finalize()* on your object when it is no longer referenced. The method is called automatically by the *garbage collector*, which reclaims unused memory.

6

Reuse in Java

A major benefit of using an object-oriented language is code reuse. By using existing class libraries and carefully designing your own classes, you reuse the code encapsulated in those classes.

In this chapter you learn about reuse in Java and are introduced to such typical object-oriented concepts as polymorphism, inheritance, aggregation, and abstraction. You also discover interfaces: Java's way of providing multiple inheritance.

Once you have learned the concepts, you practice with the familiar ATM example, using VisualAge for Java.

Reuse in Object-Oriented Languages

Reuse is one of the most compelling features of object-oriented languages in general and of Java specifically. Reuse does not mean cutting and pasting code as you would do with procedural languages like C. That approach has many limitations such as main-

tainability of bloated code. With object-oriented languages, efficient reuse is enabled by the concept of *class*. In object-oriented languages like Java you build hardly anything from scratch: you reuse already created and debugged classes from a class library. For example, if in your Java code you need to manipulate a hashtable, you do not write it from scratch. You simply use the Hashtable class from the java.util package.

In most cases, you create a variable of type Hashtable within your own classes to derive the function of the hashtable—this is called *aggregation*. Sometimes, however, you may be creating a class that is a special kind of hashtable, in which case you use *inheritance* to derive the function you need.

Is-A and Has-A Relationships

In object-oriented terminology, an inheritance relationship is often called an *Is-A* relationship. You can say that a checking account *is* a type of bank account.

A *Has-A* relationship refers to aggregation. You can say that a bank account *has* a set of transactions.

Using Is-A and Has-A is a simple way of deciding what relationships you should create.

Inheritance

Inheritance is a fundamental concept in object-oriented languages. It is a process by which classes share or reuse similar features. Just as you may have inherited certain traits from your parents, a Java class can inherit state and behavior from its parent class.

Do You Speak Object-Oriented?

First, let's get the new terms out of the way: a *parent* class, *base class*, or *superclass* is a *generalization* of a *child* class or *subclass*; a subclass is a *specialization* of a superclass or is *derived* from the superclass.

An *ancestor* class is any class higher up in the inheritance hierarchy; a *descendant* class is any class lower in the hierarchy.

Inheritance means that the specialized class adopts all properties and behavior of its ancestor class, that is, the specialized class behaves exactly like the ancestor class and owns the same attributes and methods.

These terms are all described graphically in Figure 48.

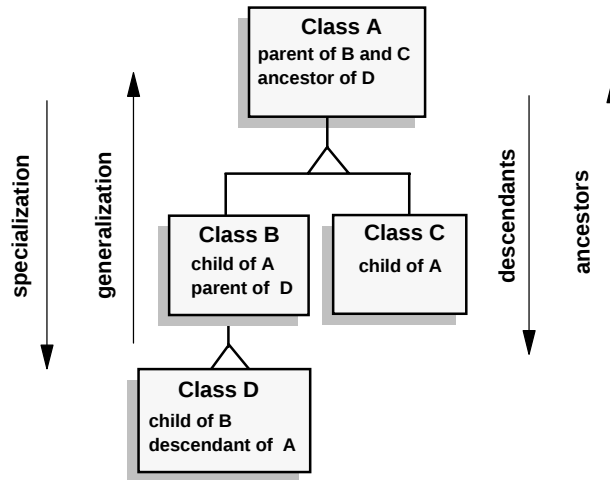


Figure 48. Inheritance Concepts

Let's have a look at Figure 49. The OverdraftAccount class is a subclass of the CheckingAccount class, and the BankAccount class is a superclass of the SavingsAccount class.

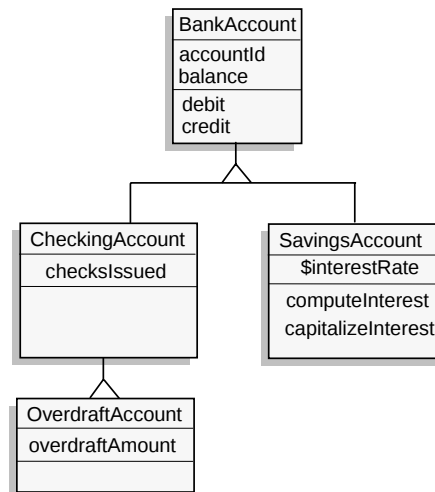


Figure 49. The BankAccount Inheritance Hierarchy

These new accounts are specializations of the existing `BankAccount` class. `CheckingAccount` has the same state and behavior plus a new field to define whether checks have been issued. The `SavingsAccount` class adds a class variable that contains the interest rate shared by all instances of `SavingsAccount` (the `$` signifies a class or static field) and two methods to calculate the interest and capitalize it. The `OverdraftAccount` class still has an `accountId`, a `balance`, and a `checksIssued` fields, you just do not have to specify it explicitly. It also has, like all other accounts, debit and credit methods that other objects can invoke.

The ability of the descendants of a class to inherit all operations of their ancestor provides a means of reusing code. Indeed your code is located in one place only. This way, you only need to modify common behavior once, inside the operations of the ancestor class.

Information



In Java all classes inherit from the *Object* class. When you declare a class without specifying its base class, using the *extends* keyword, the new class inherits directly from *Object*.

The *Object* class implements some common behavior by providing methods for all Java objects, including *toString*, *clone*, *equals*, and *finalize*.

Creating Subclasses with Java

Java uses the *extends* keyword to specify the inheritance link between two classes. For example:

```
public class CheckingAccount extends BankAccount
```

is part of the declaration for the new class, `CheckingAccount` which is a subclass of `BankAccount`. To create the `OverdraftAccount` class, you would do the same:

```
public class OverdraftAccount extends CheckingAccount
```

Incremental Development

Once a subclass is derived from a base class, it inherits all of the fields and methods of the base class. From that single fact you can reuse code already developed, but you probably want to modify specific methods to suit your needs and of course add some new fields and methods.

Method Overriding

When you replace in a subclass a method of the same signature of the base class, you *override* the base class method. Overriding means that the method called on a derived object is the method defined in the derived class, not the method inherited from the base class. For example, in Figure 49, you would override the debit method inherited by `CheckingAccount` from `BankAccount` by defining a *public void debit(double)* method in the `OverdraftAccount` class.

Sometimes you may want to reuse the method of the base class inside the method of the derived class. By using the *super* keyword, you can invoke the superclass (or base class) method from the method of the derived class. *Super* refers to the current object, but its type is the type of the superclass. Using *super* you can invoke any method and access any field of the superclass. For example, if you want to call the debit method of `CheckingAccount` from `OverdraftAccount`, you can use the *super* keyword as follows:

```
public void debit(double anAmount) {  
    super.debit(anAmount);  
    // do something else  
}
```

By invoking methods of superclasses, you can easily customize the code of your derived classes and thus program incrementally by coding only the differences between the code of the superclasses methods and the code of the derived classes.

Abstract Classes

In the real world we do not usually have generic bank accounts. We may call them bank accounts, but they are really savings accounts, checking accounts, and so on.

The same applies for the `BankAccount` class. You do not want to create an instance of a generic bank account. You create instances of `SavingsAccount` or `CheckingAccount` or `OverdraftAccount`. You only want to reuse the state and behavior of the generic bank account in its subclasses. Wouldn't it be nice to prevent developers from creating instances of the class and inform them that this class is really an abstraction of the concrete bank account classes? Well, you actually can, by making `BankAccount` an *abstract* class! When a class is defined as abstract, an instance cannot be created from it. Trying to create an instance of the class in your code leads to a compilation error. To put it a different way: There is no

BankAccount object that is not either a CheckingAccount or a SavingsAccount object. Often, classes that are not abstract are referred to as *concrete* classes because you can instantiate them.

BankAccount can be defined as abstract as follows:

```
public abstract class BankAccount {  
    // ...  
}
```

If the BankAccount class is specified as abstract, and if the debit and credit methods are declared but not defined in the BankAccount class, you must implement debit and credit methods in CheckingAccount and SavingsAccount to create instances from these subclasses.

Another example of an abstract class is Shape, representing an abstract geometric shape. It might have subclasses such as Triangle, Rectangle, or Square. The Shape class declares all common methods for these classes, such as perimeter(), surface(), draw(), or print(). The Shape class itself might implement a method such as getName() but would declare most of its methods to be abstract and only implemented by the subclasses.

An abstract class is usually used to define a common interface for a set of classes. In the Shape example, you know you can ask a Shape its perimeter. But knowing how a Circle or a Rectangle implements the perimeter method is not your problem.

Abstract Methods

Methods can also be declared as abstract. In this case they have no implementation body and must be implemented in the subclasses of the class where they are declared. Declaring methods as abstract is a design decision that forces programmers to implement the methods in their code.

For example, you can define debit() and credit() as abstract in BankAccount and force programmers who want to use BankAccount to implement these methods in their subclasses. Defining these methods as abstract could be translated in natural language as “I know that you can debit or credit a bank account, but, because each type of bank account does it differently, you must implement the methods in your concrete subclasses.”

A class containing an abstract class is called an abstract class and must be qualified itself by the *abstract* keyword. In addition, if you subclass an abstract class and you do not override all of the abstract methods of that class, your class must be declared abstract.

Polymorphism

We previously discussed overloading methods, for example writing for a class a method with the same method name but different method signatures. In contrast, an overridden method is a method that hides a method with the same signature in a superclass. Let's go back to Figure 49 and suppose for a moment that the `BankAccount` class is no longer abstract. As implemented in Chapter 3, the `debit` method of `BankAccount` checks to see that the debit amount is not greater than the balance. In the `OverdraftAccount` class you could override the `debit` method as follows:

```
public double debit(double amount)
{
    if (getBalance() + getOverdraftAmount() > amount){
        setBalance(getBalance() - amount);
    }
    else{
        System.out.println("Not enough money");
    }
}
```

First, however, you would need to make a change to the `setBalance` method. Back before you knew nothing about inheritance, you made the `balance` field and the `setBalance` method private so that the field could be changed only from within the objects of that class. Now, however, you have subclasses of `BankAccount`. They definitely want to be able to change the `balance` field too. To let them do this, you can change the access modifier on the `setBalance` method to *protected*. This is not always something that you should do; each field and method must be examined to see whether subclasses should be allowed to access it. In this case, each type of account should be able to modify the balance through the `setBalance` method.

This new `debit` method will successfully debit an account provided the debited amount is not greater than the balance combined with the overdraft limit. This method hides the `BankAccount` `debit` method: when you invoke `debit` on an object of the `OverdraftAccount` class, its `debit` method is called not the `BankAccount`'s `debit` method.

Now that `OverdraftAccount` has its own implementation of `debit`, let's consider the following code, which contains only one method:

```
public class Debiter
{
    // debitAllAccounts debits each account of an array
    // from the same amount
    public static void debitAllAccounts(BankAccount[] ba,
                                       int arrayLength,
                                       double amount) {
        for (int i = 0; i < arrayLength; i++){
            ba[i].debit(amount);
        }
    }
}
```

Beyond the fact that the subclass inherits all of the base class features, the most important aspect of inheritance is the *Is-A* relationship that links both classes. In effect, because the derived class is a type of the base class, the subclass can be used any place the base class would be. Therefore an object could call the `debitAllAccounts` method by providing an array of `BankAccount`, `CheckingAccount`, or `OverdraftAccount` objects.

If an element of the array is a reference to a `BankAccount` object, the `debit` method is called and runs the code of the `BankAccount` implementation. However, if an element of the array is a reference to an `OverdraftAccount` object, the `debit` method called is the method implemented in the `OverdraftAccount` class. In short: objects know who they are!

This propensity for objects to react differently to the same method is known as *polymorphism*, and it is one of the most essential feature of object-oriented languages.

So it goes in the real world. Remember, we do not really have generic bank accounts; we may call them bank accounts, but they are really savings accounts, checking accounts, and so on. When you debit an account, you expect the behavior of that particular account, even though you may call it a bank account. Thus, in your programs, you can treat all bank accounts generically, for example, put them into the same data structure, as we did in the previous example.

As shown in Figure 50, invoking `debit(1001)` on a `CheckingAccount` object or an `OverdraftAccount` object, both of which have a balance of \$1000, would result in very different results. The `CheckingAccount` object would refuse the debit, and the `OverdraftAccount` object would happily perform it (provided the `overdraftAmount` field is greater than \$1).

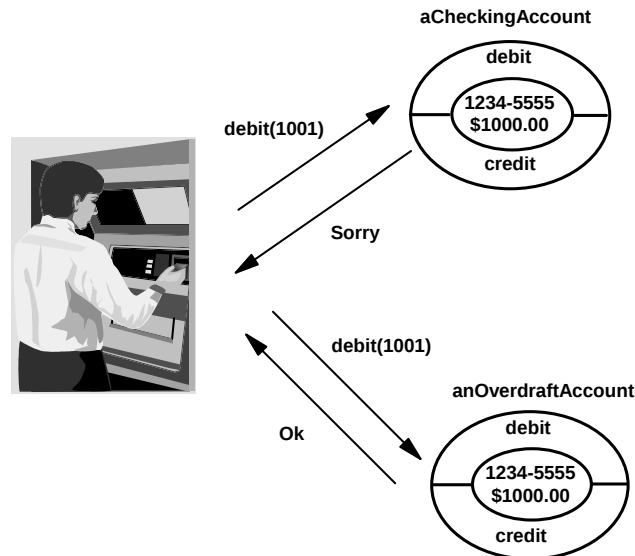


Figure 50. Invoking the Debit Method on Different Object Types

The Magic behind Polymorphism

You may wonder how Java manages to call the right method on the right object. For example, how can Java figure out that because the second element of your account array is an Over-DraftAccount object, it must call the debit method for this class? After all, it only manages a BankAccount array!

To understand the “trick” you need to learn some terminology. *Binding* is the mechanism that connects a method call to its implementation. When the compiler (and the linker in other languages like C++) performs the binding, it is called *early binding*.

In our example above, the Java compiler cannot decide which method implementation it should call for each object of the array because it does not know what the objects are. Indeed, Java knows it must manipulate an array of BankAccount references, but knowing to which object type the references are pointing is another story!

Actually, Java will know which object type the references are pointing only at run time when it calls `debitAllAccounts`. This is why Java resolves the binding at run time and why this type of binding is called *dynamic binding* or *run-time binding*.

All method binding in Java is dynamic, unless a method has been declared as final and thus cannot be overridden.

Benefits of Polymorphism

The benefits of polymorphism are multiple. First of all, polymorphism greatly increases the level of abstraction in your program. You can refer to bank accounts in your program without worrying about including all different types of bank accounts. Each time you need to debit an account, you call the debit method, not `checkingAccountDebit` or `overdraftAccountDebit`. After all, in real life you do not say, “I am going to `checkingAccountDebit` my `checkingAccount`”! Later on, if a new bank account class is defined, your code will still be up to date and will not have to undergo modifications.

Second, polymorphism dramatically improves program maintenance. If the debit method of the `OverdraftAccount` needs to be changed, you do not have to browse your code for references to the `OverdraftAccount` and make the appropriate changes. You just modify the debit method in one location: in the `OverdraftAccount` class.

Finally, polymorphism makes incremental development work. Contrary to procedural languages where you program your code by cutting and pasting code here and there, with object-oriented languages you program by difference. Once your class inherits from a class that provides 95% of the function you expect, you just have to program the 5% difference to suit your needs.

Interfaces

Interfaces are unique to the Java language. Like abstract classes, they are a way of providing a set of behaviors that other classes have to implement. Interfaces can contain abstract methods and class variables only. Interfaces do not provide reuse of behavior; rather, they provide reuse of *specification of behavior*.

Inheritance of Implementation

When a class implements an interface, it ensures users of that class that the methods declared in the interface are implemented. In other words, the interface establishes a “protocol” between classes.

Let's consider an example of an interface in the bank account scenario. Our bank would like to create special accounts for wealthy customers. These accounts would pay a higher interest rate based on the amount of money in the account. However, the bank would like to add this function only to a limited number of account types.

To implement that function, you could use the following interface:

```
public interface Profitable {
    public static final double MIN_THRESHOLD = 100000.00;
    public static final double MID_THRESHOLD = 500000.00;
    public static final double MAX_THRESHOLD = 1000000.00;

    public double bonus();
}
```

To use the interface, a class uses the *implements* keyword in the class declaration. The ProfitableAccount class implements the Profitable interface and must provide a definition for the bonus method. Any object that wants to use a ProfitableAccount object is guaranteed that the ProfitableAccount object supports the bonus method.

The ProfitableAccount class would look like this:

```
public class ProfitableAccount extends SavingsAccount
    implements Profitable {
    public double bonus() {
        double bonus = 0.0;
        if (getBalance() > ProfitableAccount.MAX_THRESHOLD)
            bonus = computeInterest() * 0.30;
        if (getBalance() > ProfitableAccount.MID_THRESHOLD)
            bonus = computeInterest() * 0.20;
        if (getBalance() > ProfitableAccount.MIN_THRESHOLD)
            bonus = computeInterest() * 0.10;
        return bonus;
    }
}
```

Notice that an interface allows only method declarations. Each method that you declare in an interface is declared *public* by default even if you do not specify this access modifier. Therefore when you implement an interface, the methods from the interface must be defined as *public*.

Apart from method declarations, interfaces can also contain data members or primitive data types. In this case they are implicitly *static* and *final*.

Variable of Type Interface

You can declare a variable of an interface, for example:

```
Profitable aProfitableAccount;
```

When you assign the variable to an instance, the instance must be of a class that implements the Profitable interface.

Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one parent class. The semantic of multiple inheritance is to add additional behavior to a class and keep all of the existing behavior. For example, you could create a new subclass of BankAccount: CheckingRetirementAccount. The new class would inherit from the CheckingAccount class and the RetirementAccount class. Thus it would have all of the features of a checking account as well as the features of a retirement account.

This all sounds very promising; however, in reality multiple inheritance must be designed very carefully. Can you write a check on a retirement account? What are the tax issues? In general the design and use of multiple inheritance are much more complicated than they are for single inheritance.

Java does not support multiple inheritance, it provides interfaces instead. In Java, a class can inherit only from one superclass, but it can implement as many interfaces as desired. Therefore while your class can have only one superclass, it can behave according to many different interfaces. Note that you must implement all methods of an interface or make your class abstract.

Interfaces provide much of the behavior of multiple inheritance without some of its problems. For example, in C++, each class you inherit from can have an implementation of the same method. In this case, your derived class ends up inheriting from several methods with the same name but from different base classes. Consequently, you must make a distinction in your code to resolve the method you want to call. In Java, you can allow multiple inheritance with interface, but only one of the classes can have an implementation, of a specific method and no ambiguity is possible.

Interfaces and Abstract Classes

Because an interface provides a way of handling multiple inheritance and can be used in the same way you would use abstract classes, you may wonder when to use an interface and when to use an abstract class in your code.

Here is the rule of thumb: Because an interface provides the abstract class function and a way of upcasting to more than one base class, use an interface if you can create a base class without method definition or instance variables. Otherwise, if you have to implement a method or define an instance variable in your class, use an abstract class.

Aggregation

Aggregation is the reuse of classes by placing them inside the classes you build. It is a good way of building very complex objects. By including a class as an instance variable in the class you are building, you have access to all of the public behavior of that included class.

Banking Transactions

In many cases you should consider aggregation before inheritance. For example, let's say you want your bank accounts to create a record for every transaction they perform. You could subtype from some sort of transaction record class, if it existed, to get the behavior needed, but you are already inheriting from the BankAccount class. You could implement a Transaction interface, but you would have to write the transaction code for every class that wanted to record these transactions.

Another way of providing this transaction behavior is to aggregate your transaction objects within the BankAccount class. Figure 51 is an object model representing the aggregation of the transactions within a bank account at the analysis stage. Remember that in the analysis stage you do not worry about the design and implementation of the system.

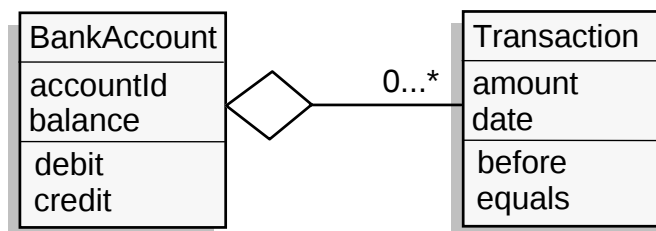


Figure 51. UML Representation of Aggregation

The left box represents the `BankAccount` class. The right box represents the `Transaction` class. The relationship between the two is an *aggregation* and is represented by a diamond connected to a line which joins the two classes. On the other end of the connection is the *multiplicity*: the number of aggregated elements. In this example, the relationship expresses that the bank account has zero or more transactions.

The design and implementation of an object model often differ from the object model in the analysis phase. In this case our analysis determined that our bank account should contain a number of transactions. To implement this aggregation of several transactions, you could use an array or the Java `Vector` class from the `java.util` package. In this example you use a `Vector` because of its dynamic size capabilities.

```
import java.util.*;

public class BankAccount {
    private Vector log;
    private String accountId;
    protected double balance;
    private static double initialAmount = 20.0;

    public BankAccount(String anID) {
        log = new Vector();
        accountId = anID;
        balance = initialAmount;
    }
    // rest of the code ...
}
```

Information



The `Vector` class supports an array of objects that can change in size. You can access objects contained in the `Vector` can be accessed through an integer index or by iterating through the `Vector`, using an enumeration. For more information about the `Vector` class, see the [JDK documentation](#).

The `BankAccount` class contains one instance variable `log` of type `Vector`. The `log` represents the aggregation shown in the previous code. To make the transaction log functional, we create methods that add transactions to the log sorted by date.

The log holds instances of the Transaction class defined as follows:

```
import java.util.*;

public class Transaction
    private Date date;
    private double amount;
}
```

A before method in the Transaction class provides a way of comparing transactions. The before method takes one argument and returns true if the receiver's (the object whose before method was invoked) date is before the argument's date.

```
public final boolean before(Transaction other) {
    return date.before( other.getDate());
}
```

Delegation

In the before method, the Transaction class gives responsibility to the Date class to perform the comparison. The term *delegation* is used in this case to indicate that the Transaction class delegates the comparison to the Date class.

Notice the use of the *final* modifier. Methods that are not designed to be overridden should be declared as final. This declaration will improve the performance of your programs and make it clear that the method is to be used, not overridden.

To add transactions to the log, you must introduce a method called addTransaction in the BankAccount class. The parameter, a Transaction object, is added at the correct position in the log:

```
void addTransaction(Transaction aTrans) {
    boolean found = false;
    int i = 0, logSize = log.size();
    Transaction trans = null;
    for(i=0; i< logSize && !found; i++) {
        trans = (Transaction) log.elementAt(i);
        if (aTrans.before(trans)) {
            found = true;
            log.insertElementAt(aTrans, i);
        }
    }
    if (!found)
        log.addElement(aTrans);
}
```

Notice that the `addTransaction` method has no access modifier. Lack of an access modifier implies package level access: Other classes within the package can invoke this method, but it cannot be invoked by other classes, even if they are subclasses, but in another package.

Association

Another relationship between objects is *association*. In Java association and aggregation are both implemented through object references. To understand the difference between the two relationships, look at where the referenced object is created. If the containing object instantiates or initially creates the object in some other way, the relationship is usually aggregation. If the containing object assigns the object a reference from an existing reference, the relationship is an association.

Inheritance and Aggregation

Both inheritance and aggregation enable you to achieve reuse with Java, and you may wonder which to choose.

Use inheritance when you want to specialize and distinguish a class from an already existing one. If you create a subclass from a base class, users of the new subclass have access to the interface of the base class as well as the interface of the new class.

Use aggregation when you want to add features of an existing class to a new class but not its interface. By embedding an object in your new class, you do not necessarily expose its interface. This is especially true if you embed private objects in your class.

Inner Classes

With JDK 1.1 and higher it is possible to define classes inside of other classes. The inside classes are called *inner* classes. By defining a class inside of another, you can group classes that logically should be together and control the visibility of one within the other. However do not confuse inner classes with aggregation.

As you will see in Handling Events with Java AWT 1.1 on page 208, inner classes can be useful for creating clean code for handling user events by separating the GUI of your application from its logic.

In its current version, VisualAge for Java does not support inner classes. It is not possible to import Java source code containing inner classes in your repository. However, you can import a class file compiled from the Java compiler even if the source code of its class contains inner classes. Note that, because you import the .class file, you cannot browse the source code of the imported class.

For more information about inner classes, refer to the JDK documentation.

Class Implementation Revisited

Inheritance has an impact on several Java concepts. In this section, we review those concepts in the context of your new knowledge.

Modifiers Revisited

In Chapter 4 and Chapter 5, we discuss modifiers—but not all of them, and some of them probably did not make a lot of sense, because you did not have the object-oriented knowledge to fully understand all the concepts. Below, we list the modifiers for classes, methods, and fields.

Class Modifiers

Access Modifiers

public	The object can be accessed by any other object.
<i>no modifier</i>	Objects of the class can be accessed by any object in the same package (package access).

Other Modifiers

abstract	This class cannot be instantiated.
final	This class cannot be subclassed.

Method Modifiers**Access Modifiers**

public	The method can be invoked by any other object.
protected	The method can be invoked by any object in the same package or any subtype of this class.
<i>no modifier</i>	The method can be invoked by any object in the same package (package access).
private	This method can be invoked only within objects of this class.

Other Modifiers

static	This is a class method and can access static fields only.
abstract	This method has no implementation in this class and must be overridden in a subtype.
final	This method cannot be overridden.

Field Modifiers**Access Modifiers**

public	The field can be accessed by any other object.
protected	The field can be accessed by any object in the same package or any subtype of this class.
<i>no modifier</i>	The field can be accessed by any object in the same package (package access).
private	This field can only be accessed within this object.

For the other field modifiers, see Variables on page 21.

Constructors Revisited

When you create an instance of a class that inherits from a superclass, Java automatically calls the superclass default constructor if the constructor of the subclass is not overloaded.

If the constructor of the subclass is overloaded, the default constructor no longer exists, and the programmer must write one (if necessary). If the subclass does not have a default constructor, the base class constructor must be called from the subclass constructor by using the *super* keyword as described in Method Overriding on page 107.

Reuse with VisualAge for Java

In this section we reinforce the concepts you learned in this chapter by practicing using VisualAge for Java.

Copy your existing example to a new package. Select the ch05 package and then click on **Selected**→**Copy**. Enter ch06 as the new name for the package.

Implementing Inheritance

In this section you implement inheritance by creating the subclasses of the BankAccount: CheckingAccount and SavingsAccount. Here is an overview of the exercise:

1. Create two new subclasses of BankAccount: CheckingAccount and SavingsAccount.
2. Add a checksIssued field to the CheckingAccount class and implement the getter and setter for it.
3. Add two methods (computeInterest and capitalizeInterest) and a class variable (static field) to SavingsAccount.
4. Add a new subclass of CheckingAccount: OverdraftAccount.
5. Add a new double field, overdraftAmount, to the OverdraftAccount class.
6. Override the debit method in OverdraftAccount to use the new overdraft information.
7. Test your work.

Modifying the BankAccount Class

In order for the subclasses of BankAccount to access the balance field, you must change the access modifier of the setBalance method to *protected*. You will also want to override the toString method from the Object class.

1. Select the `setBalance(double)` method in `ch06.BankAccount`. In the Source pane, change the private access modifier on `setBalance` to *protected*.
2. Create a `toString` method:

```
public String toString()
{
    java.text.NumberFormat f =
        java.text.NumberFormat.getCurrencyInstance();
    String str = getClass().getName() + " #" +
        getAccountId() + "\t\tBalance:" +
        f.format( getBalance());
    return str;
}
```

There are two new items in the `toString` method:

- The `getClass().getName()` method. Java provides powerful mechanisms for accessing information about objects. In this case you can print the actual name of the class.
 - The `java.text.NumberFormat.getCurrencyInstance()` method. This method allows you to format currency amounts for display. Using the default locale (English - United States in our case) automatically truncates the currency amounts to two decimal places, shows negative amounts in parentheses and places a \$ before the amount.
3. Remove the `finalize` method in the `BankAccount` class; otherwise “Bye for Now” will print out on the console each time a `BankAccount` object goes out of scope. Select the method, then click on **Selected**→**Delete**.

Creating the CheckingAccount Class

1. In the Workbench select `BankAccount` in the `ch06` package.
2. Select the **New Class/Interface** menu item from the Selected menu bar or the context menu. The SmartGuide - Create Class or Interface opens. The Project, Package and Superclass fields should already be filled in (Figure 52).

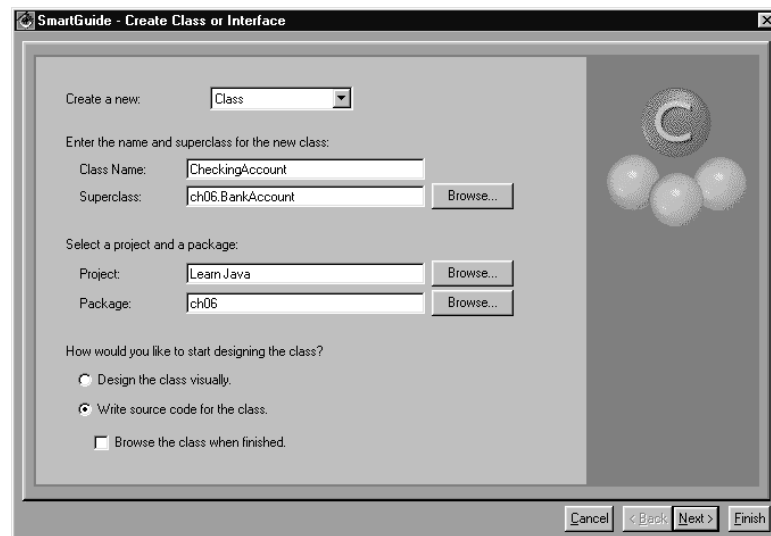


Figure 52. Creating the CheckingAccount class

3. Type **CheckingAccount** in the Class Name field, select the **Write source code for the class** radio button, select the **Browse the class when finished** checkbox and click **Next**.
4. Make sure the **Copy constructors from superclass** checkbox is selected and click **Finish**. If you do not select the checkbox, you receive an error, because the default constructor of the new class would be attempting to call `super()`; the default constructor of the BankAccount class that is no longer accessible.
5. Add a new private boolean field to the CheckingAccount class, call it `checksIssued`, and provide public final getter and setter methods for it (Figure 53):


```
public final boolean getChecksIssued()
public final void setChecksIssued( boolean checksIssuedParam)
```
6. You can create new fields by simply typing the code into the class declaration in any Source pane.

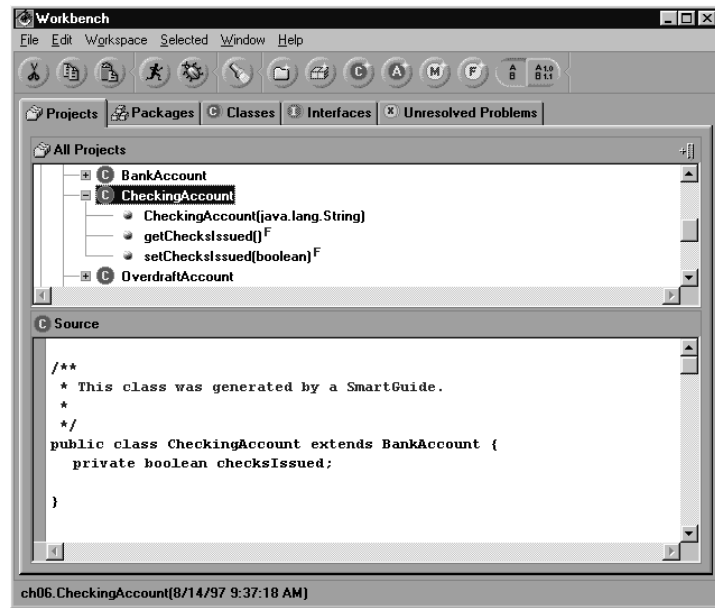


Figure 53. The CheckingAccount Class in the Workbench

Creating the SavingsAccount Class

1. Create a new subclass of BankAccount and name it SavingsAccount. Repeat steps 2 through 4 of Creating the CheckingAccount Class on page 122, substituting SavingsAccount for CheckingAccount.
2. Add a new class variable (static field) to the new class. It should be a *private static double* field named *interestRate* with an initial value of 3.0. Provide *static* getter and setter methods for it (Figure 54):

```
public final static double getInterestRate() {
    return interestRate;
}

public final static void setInterestRate(double aRate) {
    interestRate = aRate;
}
```

3. Create two public methods: one called *computeInterest*, returning a double, and another called *capitalizeInterest*, returning void. Their access should be public. Do not make *capitalizeInterest* final because you will be subclassing it later. Implement the two public methods, using the following code (Figure 54):

```

public final double computeInterest() {
    return getBalance() * getInterestRate();
}

public void capitalizeInterest() {
    setBalance( getBalance() + computeInterest());
}

```

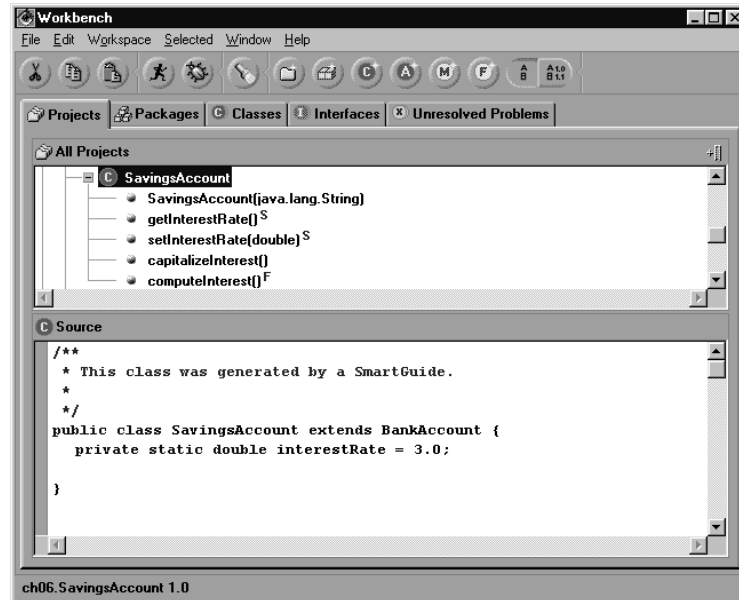


Figure 54. The SavingsAccount Class in the Workbench

Creating the OverdraftAccount Class

To implement the OverdraftAccount class, you must first modify the BankAccount class. You could create the OverdraftAccount class and override the debit method, however; you would have to change two debit methods if you want to change the logic involved in a debit.

A better solution is to add another method on BankAccount: `getAvailableFunds`. This method typically returns the balance. For the OverdraftAccount, however, it returns the balance and the overdraftAmount.

1. Create the new method for the `ch06.BankAccount` class:

```

public double getAvailableFunds()
{
    return getBalance();
}

```

2. Update the debit function to use the new method:

```
public void debit( double anAmount)
{
    if( anAmount > getAvailableFunds()){
        System.out.println("Sorry, not enough money");
    }
    else{
        setBalance( getBalance() - anAmount);
    }
}
```

3. Create a new subclass of CheckingAccount and name it *OverdraftAccount*. Repeat steps 2 through 4 of Creating the CheckingAccount Class on page 122.4. Add a new private double field to the class, call it *overdraftAmount*, and provide public final getter and setter methods for it.

5. Override the getAvailableFunds method by creating a new getAvailableFunds method on this class (Figure 55):

```
public final double getAvailableFunds()
{
    return getBalance() + getOverdraftAmount();
}
```

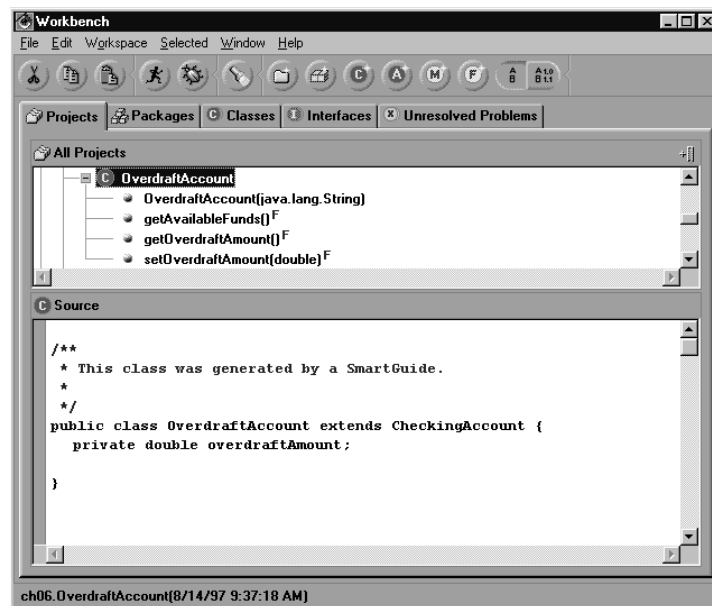


Figure 55. The OverdraftAccount Class in the Workbench

Testing Your Work

To test your new classes, create a class called *BankAccountTest*. There are several benefits of using a dedicated class rather than the Scrapbook for testing:

- ☐ You can embed the class in a larger test suite.
- ☐ You can run the test outside the VisualAge for Java environment.

To create the class:

1. Select the `ch06` package and then select **Selected**→**New Class/Interface**. Enter **BankAccountTest** for the *Class Name*.
2. Click **Next** and select the **main(String[])** checkbox, then click **Finish**.
3. Add the following code to the new main method:

```
java.text.NumberFormat f =
    java.text.NumberFormat.getCurrencyInstance();
BankAccount accnt = new BankAccount("1234-5678");
System.out.println("BankAccount Subclasses Example\n");
accnt.credit(10000.00);
accnt.debit(1234.00);
accnt.debit(344.00);
System.out.println(accnt);
accnt = new CheckingAccount( "1234-5679");
accnt.credit(1000.00);
accnt.debit(1234.00);
System.out.println(accnt);

OverdraftAccount overdraft =
    new OverdraftAccount( "1234-5676");
overdraft.setOverdraftAmount( 1000.00);
overdraft.credit(1000.00);
overdraft.debit(1234.00);
System.out.println( overdraft);
SavingsAccount saving = new SavingsAccount( "1234-5677");
saving.setInterestRate(0.065);
saving.credit( 10000.00);
saving.debit( 1234.00);
saving.debit( 344.00);
System.out.println( "Interest for account #" +
    saving.getAccountId() + " is: " +
    f.format( saving.computeInterest()));
saving.capitalizeInterest();
System.out.println(saving);
```

4. Now select the `BankAccountTest` class and use the **Run**→**Run main** menu item in the **Selected** menu; click **Run** in the dialog window that is displayed. In the Console window, you should get the results shown in Figure 56.

Notice the declarations in the `BankAccountTest` class. You can assign a `BankAccount` variable to an object that is a subtype of the `BankAccount` class. However, you can only use the `BankAccount` variable to access the methods declared in the `BankAccount` class. To invoke the `computeInterest`, `capitalizeInterest`, `setInterestRate`, and `setOverdraftAmount` methods, you must use the appropriate variable type.

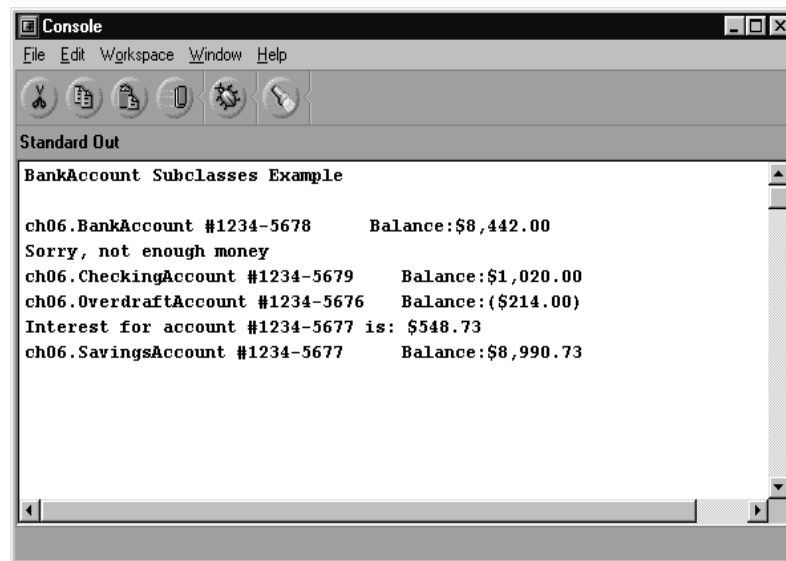


Figure 56. Running the Subclasses Example

Implementing Abstract Classes

In this section you make `BankAccount` an abstract class. Then you test the new code, using a modified `BankAccountTest` class.

Making the BankAccount Abstract

Follow these steps to make the `BankAccount` class abstract:

1. Select the `ch06.BankAccount` class in the Workbench.
2. In the Source pane, add the *abstract* keyword class definition:

```
public abstract class BankAccount
```

- Now you should see a grey cross next to the `BankAccountTest` class and a red cross next to its `main` method. Select the main method and check the message on the status line. It should read:

Cannot create an instance of the abstract class ch06.BankAccount

- What's wrong? You cannot create an instance of the abstract `BankAccount` class! Correct the `BankAccountTest` class by removing the code in the main method that uses the `BankAccount` instance. The code to remove is shown below in italics. Then add the `BankAccount acctnt` declaration shown in bold.

```
public static void main( String[] args) {
    java.text.NumberFormat f =
        java.text.NumberFormat.getCurrencyInstance();
    BankAccount acctnt = new BankAccount("1234-5678");
    System.out.println("BankAccount Subclasses Example\n");
    acctnt.credit(10000.00);
    acctnt.debit(1234.00);
    acctnt.debit(344.00);
    System.out.println(acctnt);

    BankAccount acctnt;
```

The modified main method should look like this:

```
public static void main( String[] args) {
    java.text.NumberFormat f =
        java.text.NumberFormat.getCurrencyInstance();
    System.out.println("BankAccount Abstract Class Example\n");
    BankAccount acctnt;
    acctnt = new CheckingAccount("1234-5679");
    acctnt.credit(1000.00);
    acctnt.debit(1234.00);
    System.out.println(acctnt);

    OverdraftAccount overdraft =
        new OverdraftAccount("1234-5676");
    overdraft.setOverdraftAmount(1000.00);
    overdraft.credit(1000.00);
    overdraft.debit(1234.00);
    System.out.println( overdraft);
    SavingsAccount saving = new SavingsAccount("1234-5677");
    saving.setInterestRate(0.065);
    saving.credit(10000.00);
    saving.debit(1234.00);
    saving.debit(344.00);
    System.out.println("Interest for account #" +
        saving.getAccountId() + " is: " +
```

```

        f.format( saving.computeInterest()));
    saving.capitalizeInterest();
    System.out.println(saving);
}

```

- Now run your modified BankAccountTest class (Figure 57). Notice the different results? This time you did not credit the checking account with quite as much money, so the debit failed.

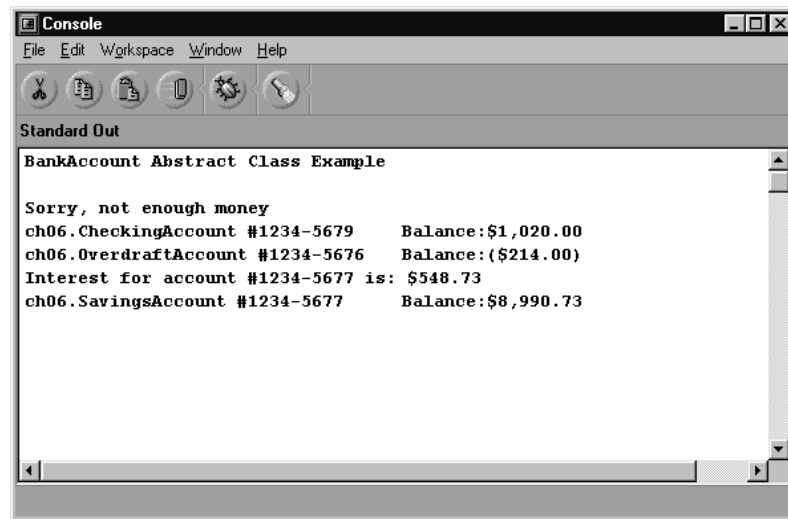


Figure 57. Running the Abstract Class Example

Implementing an Interface

In this section you create the Profitable interface. The Profitable interface will be implemented by a new subclass of SavingsAccount called ProfitableAccount. Then you modify the class to test the changes.

Creating the Profitable interface

- Select the ch06 package.
- Create a new interface by selecting **Selected→New Class/Interface**. Select **Interface** in the Create a new: drop-down list.
- Enter **Profitable** in the Interface Name field. Click **Finish** to create the interface.
- Select the new interface in the Workbench and then click the **Create method or constructor** tool bar button.

5. Enter **double bonus()** in the Method Name field. Notice that all the checkboxes and radio buttons are disabled because you are creating an interface. Click **Finish**.
6. Select the Profitable interface again and add the following attributes to the declaration (Figure 58):

```
public static final double MIN_THRESHOLD = 100000.00;
public static final double MID_THRESHOLD = 500000.00;
public static final double MAX_THRESHOLD = 1000000.00;
```

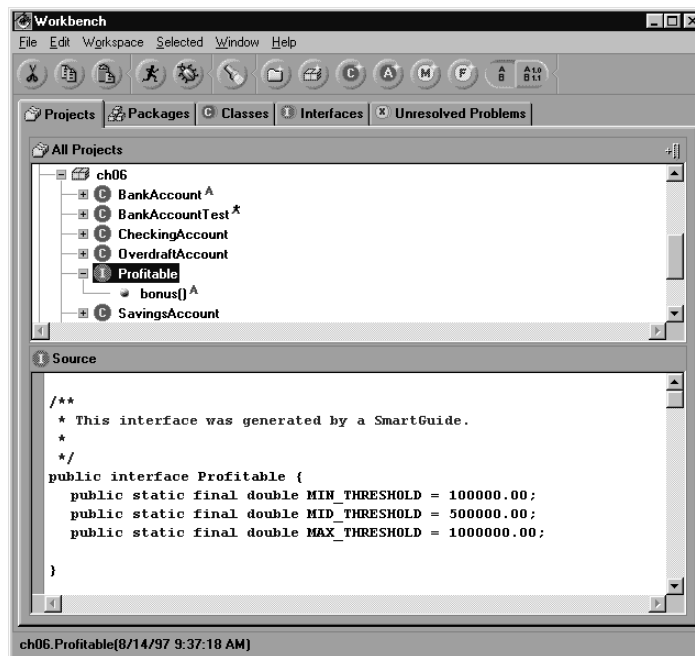


Figure 58. The Profitable Interface in the Workbench

Implementing the Profitable Interface

1. Select the ch06.SavingsAccount class.
2. Add a new subclass, using **Selected→New Class/Interface**. Fill in the name of the class with: **ProfitableAccount**.
3. Click **Next>** and make sure that the **Methods which must be implemented** and the **Copy constructors from super-class** checkboxes in the *Which method stubs would you like to create?* group are selected. VisualAge for Java creates the method stubs for the bonus method and for the constructor.
4. Click **Next>** and then **Add...** to add the Profitable interface (Figure 59). Click **Finish** to create the class. Notice that all possible interfaces starting with the letters you type are shown.

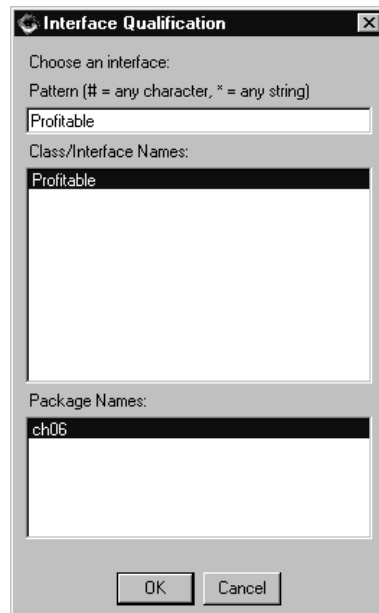


Figure 59. Selecting the Profitable Interface

5. Modify the stub of the bonus method:

```
public double bonus() {
    double theBonus = 0.0;

    if ( getBalance() > ProfitableAccount.MAX_THRESHOLD)
        theBonus = computeInterest() * 0.30;
    else if ( getBalance() > ProfitableAccount.MID_THRESHOLD)
        theBonus = computeInterest() * 0.20;
    else if ( getBalance() > ProfitableAccount.MIN_THRESHOLD)
        theBonus = computeInterest() * 0.10;

    return theBonus;
}
```

6. Create a capitalizeInterest method in the ProfitableAccount class that overrides the method in the SavingsAccount class:

```
public void capitalizeInterest() {
    setBalance( getBalance() + computeInterest() + bonus());
}
```

Testing the ProfitableAccount Class

1. Modify the main method of BankAccountTest :

```
public static void main( String[] args) {
    java.text.NumberFormat f =
```

```

        java.text.NumberFormat.getCurrencyInstance();
        ProfitableAccount acct =
            new ProfitableAccount("1234-5678");
        acct.setInterestRate( 0.1);
        System.out.println("ProfitableAccount Example\n");
        acct.credit(10000.00);
        acct.debit(1234.00);
        acct.debit(344.00);
        acct.credit(1000000.00);
        System.out.println("Interest for account #"
            + acct.getAccountId()
            + " is " + f.format( acct.computeInterest()) + "\n");
        System.out.println("Bonus for this Profitable account: "
            + f.format( acct.bonus()) + "\n");
        acct.capitalizeInterest();
        System.out.println(acct);
    }

```

2. Run the class (Figure 60).

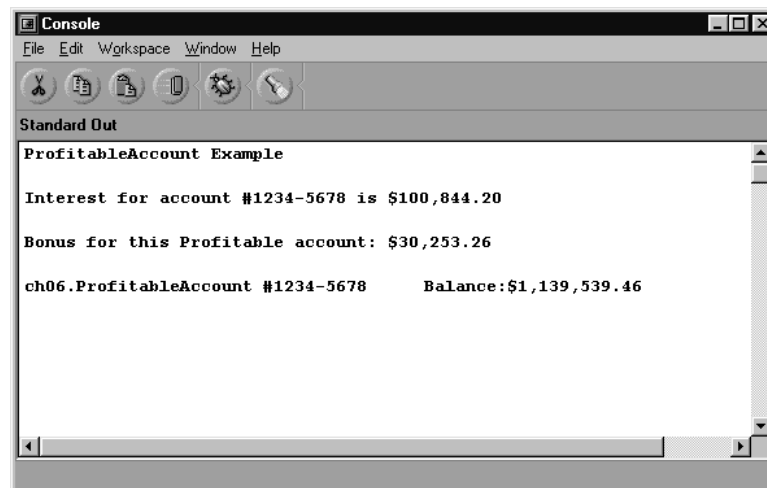


Figure 60. Running the ProfitableAccount Example

Implementing Aggregation

In this section you create a *Transaction* class that imports the *java.util* package. Then you modify the *BankAccount* class to hold the transactions created.

Creating the Transaction Class

1. Create a new class named *Transaction* in the *ch06* package, as shown on Figure 61.

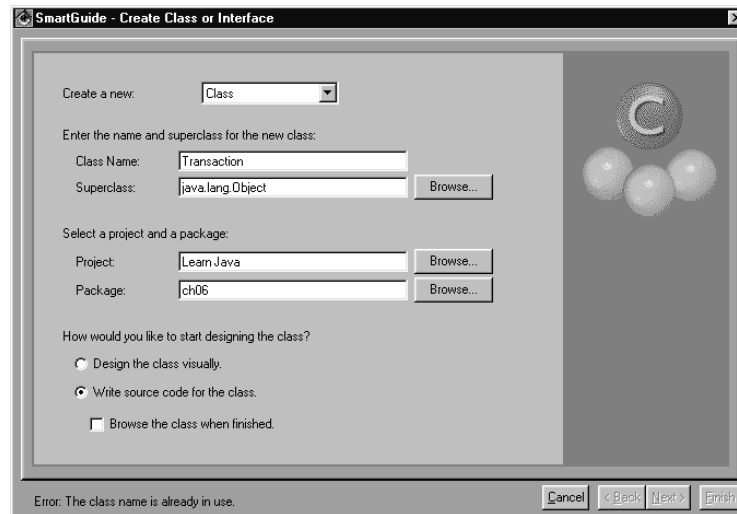


Figure 61. Creating the Transaction Class

2. Click **Next>** and check the **toString()** checkbox.
3. Import the java.util package by clicking the **Add Package...** button, selecting **java.util**, and clicking **Finish**.
4. Add two new private fields to the Transaction class:
 - A field *date* of type Date (from the java.util package):

```
private Date date;
```
 - A field *amount* of type double

```
private double amount;
```
5. Provide public final getters and setters for these fields:


```
public final Date getDate()
public final void setDate(Date aDate)
public final double getAmount()
public final void setAmount(double anAmount)
```
6. Create a constructor with two parameters: a Date and a double. Use these parameters to initialize the corresponding fields:


```
public Transaction(Date aDate, double anAmount) {
    date = aDate;
    amount = anAmount;
}
```
7. Add a method called *before* in the Transaction class to sort the transactions:

```

    public final boolean before(Transaction other) {
        return date.before( other.getDate());
    }

```

8. Implement the *toString* method, using the following code:

```

public final String toString() {
    java.text.NumberFormat f =
        java.text.NumberFormat.getCurrencyInstance();
    return "Transaction Date:" + date.toString()
        + "\t\tAmount: " + f.format( amount);
}

```

You can delete any originally generated code in the *toString* method. The *Transaction* class should display in the Workbench as shown in Figure 62.

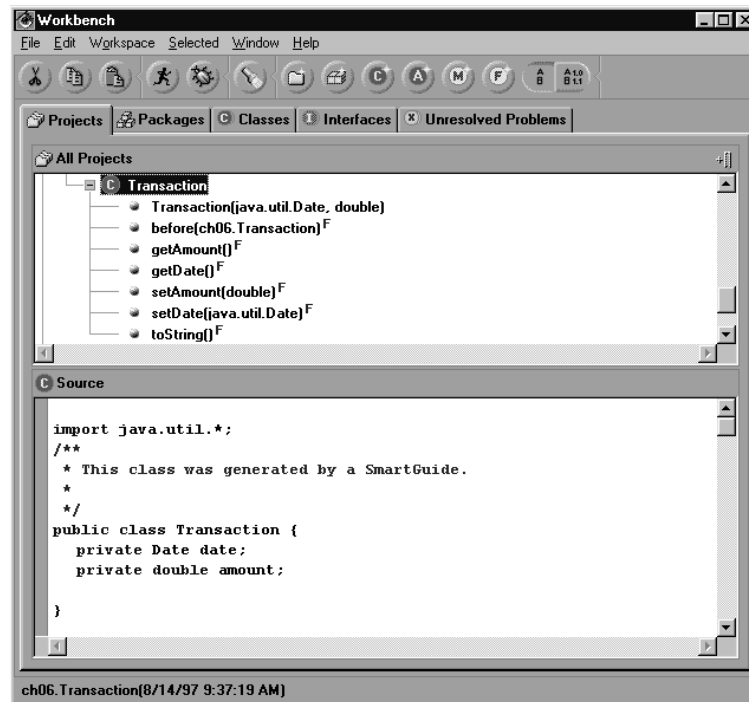


Figure 62. The *Transaction* Class in the Workbench

Modifying the BankAccount Class

Now you have to make some changes to the *BankAccount* class to support the transactions:

1. Add a new private field named, *log*, of type *Vector* (from the *java.util* package) to the *BankAccount* class. Use the import *java.util.**; statement in the *BankAccount* declaration
2. Modify the existing constructor of the *BankAccount* class to initialize the *log* field with a new *Vector* object. Use the following code as a guide:

```
public BankAccount(String anAccountId) {  
    accountId = anAccountId;  
    balance = initialAmount;  
    log = new Vector();  
}
```

3. Create a new method, in the *BankAccount* class, called *addTransaction*, which will add a transaction to the bank account's log, using the following code:

```
void addTransaction(Transaction aTrans) {  
    boolean found = false;  
    int i = 0, logSize = log.size();  
    Transaction trans = null;  
    for(i=0; i < logSize && !found; i++) {  
        trans = (Transaction) log.elementAt(i);  
        if (aTrans.before(trans)) {  
            found = true;  
            log.insertElementAt(aTrans, i);  
        }  
    }  
    if (!found)  
        log.addElement(aTrans);  
}
```

4. Now create overloaded versions of *credit* and *debit*, which enable you to perform credit and debit transactions on specific dates and which add the transaction to the bank account's log. Notice the changes in bold:

```
public void credit( Date aDate, double anAmount) {  
    setBalance( getBalance() + anAmount);  
    addTransaction(new Transaction( aDate, anAmount));  
}  
public void debit( Date aDate, double anAmount)  
{  
    if( anAmount > getAvailableFunds()){  
        System.out.println("Sorry, not enough money");  
    }  
    else{  
        setBalance( getBalance() - anAmount);  
        addTransaction(new Transaction( aDate, -anAmount));  
    }  
}
```

5. Now change the original debit and credit methods to make full use of the object-oriented capabilities of Java. The original debit and credit methods call the new methods with the current date. Later, if you want to change the implementation of debit or credit, you only need to change it once within the class. The debit and credit changes are:

```
public void debit(double anAmount)
{
    debit(new Date(), anAmount);
}

public void credit(double anAmount) {
    credit(new Date(), anAmount);
}
```

6. Modify the toString method:

```
public String toString()
{
    java.text.NumberFormat f =
        java.text.NumberFormat.getCurrencyInstance();
    String str = getClass().getName() + " #" +
        getAccountId() + "\t\tBalance:" +
        f.format( getBalance());
    Transaction trans = null;
    for(Enumeration enum = log.elements();
        enum.hasMoreElements();) {
        trans = (Transaction)enum.nextElement();
        str += "\n\t" + trans.toString();
    }
    return str;
}
```

7. Change the main method in BankAccountTest to the following (you have to import the java.util.Date class into the BankAccountTest class):

```
public static void main( String argv[]) {
    CheckingAccount acct =
        new CheckingAccount("1234-5678");
    System.out.println("Aggregation Example\n");
    acct.credit( new Date("07-mar-97"), 10000.00);
    acct.debit( new Date("23-jan-97"), 1234.00);
    acct.debit( new Date("01-dec-91"), 344.00);
    System.out.println(acct);
}
```

8. Run the BankAccountTest class and verify that the transactions are sorted correctly (Figure 63).

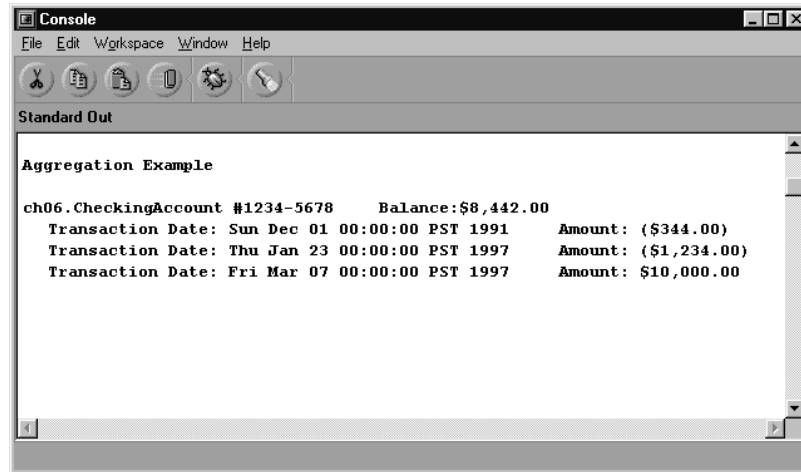


Figure 63. Running the Aggregation Example

Summary



In this chapter, you learned about code reuse in Java through:

- ❑ Inheritance: Subclasses inherit both state and behavior from their ancestors. Methods can be overridden and overloaded.
- ❑ Abstract classes, which define behavior and state for subclasses but are never instantiated.
- ❑ Interfaces, which provide different types of behaviors for classes.
- ❑ Aggregation, which supports reuse by aggregating objects within other objects.

7

Error Handling and Debugging

Although Java is designed from the ground up to help you write clean and well-designed programs, errors might sneak into your code. In this chapter you learn how to use the powerful Java exception mechanism to help detect and report errors in your programs. To reinforce the concepts that are introduced, you enhance your bank account classes with exception handling features and use the VisualAge for Java Debugger and Inspectors to diagnose and fix problems in your code.

Overview of Exception Handling in Java

In this section we explain the need for a new mechanism for handling errors in programs, what exceptions are, and how you can use exceptions to create alternative flow in your programs to handle unexpected error conditions.

Why a New Error Handling Mechanism?

In the old days programmers relied on the return code mechanism to handle errors in their programs. Simply stated, they wrote functions that would return a status code to be interpreted by the caller functions. If a status code reported a run-time error, the caller would react accordingly by displaying an error message or executing an error handling code.

Handling errors by means of the return code is not suitable for mission-critical applications for several reasons:

Bug-free program syndrome: Many programmers believe that the code they write is rock solid and they do not need to check the return code they get by calling library functions. For example, when passing incorrect arguments to a function, they may get erroneous results that can be propagated along their code if they do not check the error condition returned by the function.

Program complexity: Programmers need to write alternative code to handle errors each time they call a function. Consequently, code complexity grows, and programs get bloated by error handling code that tend, to be bigger than the actual code that gets the work done.

Limited information: Most of the time, a return code transmitted by a function to a caller is an integer value. Thus it carries limited information and must be interpreted by some extra conversion code written in the caller. For example, the debit method of the `CheckingAccount` class could return 0 if the debit method is completed with no errors, 1 if the debit cannot be completed for insufficient balance reasons, 2 if the debit amount is negative, and so on. In addition, the conversion code that is written for the debit method may not be suitable for another method called by the same caller. Moreover, some methods such as constructors or methods declared with a void return type do not return any return code!

Return code standardization: Although return codes are often integer values, it is very difficult to get a standard in the way the error condition is coded. In addition, the code can differ greatly from one library to another.

Needless to say, it is difficult to write solid code by relying on return codes for error handling. A different approach is necessary. The approach should force programmers to write error handling code and check for condition errors. It should also facilitate program maintenance by clearly defining two program flows: one flow to handle normal execution, and one flow to handle error conditions.

Since JDK 1.0, Java has proposed such an approach, which is based on exceptions and the delegation model.

What Are Exceptions?

Exceptions are events that take place when exceptional conditions occur during program execution. Exceptions can be caused by defective hardware, such as memory or disk errors; programming errors, such as writing beyond an array boundary; or violating the defined use of an object, for example, not providing a valid password to log on to a database. In all these cases an exception can be signaled or *thrown*.

In Java, exceptions are objects that contain useful information for recovering from the exception condition. Figure 64 shows the flow of an exceptional situation. Fred is trying to withdraw \$100 from his bank account at an ATM. The account indicates, or throws, an exception: “Sorry, you don’t have enough money.” Fred can then choose to withdraw only the amount he actually has in the account.

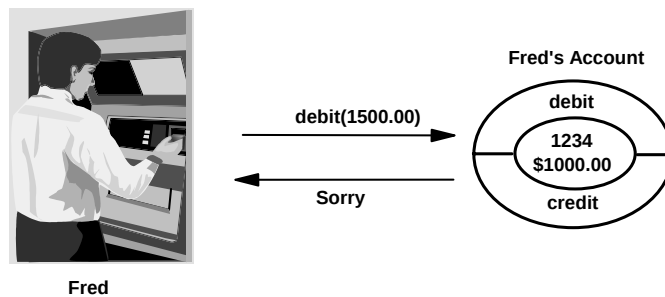


Figure 64. An Exception Condition

Exception Handling Keywords and Concepts

In the example in Figure 64, the account throws an exception that Fred receives. At this point Fred realizes that the account has an insufficient balance and that he should handle the exception. Fred can either handle the exception by himself or *delegate* someone to handle the exception. Similarly in your program, you can receive an exception and decide to handle the exception by using a specific statement block, called a *catch* block, or delegating another object to handle the exception by *rethrowing* it.

Throwing an Exception

Before catching an exception, it must be thrown by an instance or class method. In Java, you need to inform the caller of a method of the exceptions that might be thrown out of that method. In this way, the caller knows the code it must write to catch all potential exceptions. Of course, you could look up the method code to check for throw statements, but often, if you use a class library, the source code is not available and you need a different approach.

In Java, methods specify exceptions they might throw by using the **throws** keyword in their definition:

```
void debit(double anAmount) throws InsufficientFundsException {  
    // Some code ....  
}
```

In this example, the debit method declares that it might throw an `InsufficientFundsException` in certain conditions. When the exception occurs, an object of type `InsufficientFundsException` is created from the heap using the new method. Then, the current path of execution is stopped, and the exception object is transmitted to an exception handler code, which takes over.

Catching an Exception

To catch an exception in your code, you place methods that can throw exceptions within a *try* block. In Figure 64, we assume that the debit method of the `myAccount` object can throw an *InsufficientFundsException* so it is placed within a *try* block:

```
try {  
    myAccount.debit(100.00);  
}
```

You then indicate your willingness to handle exceptional situations by defining *catch* blocks. Each catch block specifies the exception type in which the caller is interested. The code inside the catch block is commonly called the *exception handler*. Here is a catch block you could use for the example:

```
catch (InsufficientFundsException e) {  
    myAccount.debit( myAccount.getBalance());  
}
```

If any code inside a *try* block throws an exception of the specified class (or its subclasses) in the catch block, the program branches immediately to the catch block. When you define *try* and *catch*

blocks, your program has two execution flows: The try block lists the code for a normal execution, and the catch block lists the code for an abnormal execution.

A complete try-catch block would look like this:

```
void getMoneyFromATM(BankAccount account, double amount)
{
    try {
        account.debit(amount);
    }
    catch(InsufficientFundsException e) {
        account.debit(account.getBalance());
    }
}
```

If the exception thrown in a method try block cannot be processed by the corresponding catch block or if a catch block is not defined, the exception is passed to the method that called the method in the next-higher context, and so forth ,recursively.

The Finally Block

Whether or not an exception occurs, you might want to execute a piece of code to clean up resources allocated in a try block. For this purpose Java provides you with an optional *finally* block, which plays a similar role to the finalizer method of your classes:

```
try {
    // code that runs under the control of an exception handler
} catch (aExceptionClass e) {
    // handler code that handles the exception
} finally {
    // some clean up code
}
```

Let's see what is happening in the above code snippet:

- ❑ First, Java executes the code defined in the try block.
- ❑ If an exception is thrown in the try block, Java tries to match the exception with the type specified in the catch block. If Java finds a match, it executes the catch block.
- ❑ Whether or not an exception was thrown, the statements within the finally block are executed.

Catching Multiple Exceptions

It is possible to catch several exception types by using cascaded catch blocks in a try-catch structure. For example, imagine that you have to do some sophisticated exception handling during file I/O operations and you want to catch different exceptions that could be thrown by the Java I/O classes. Your code would look like:

```
try {
    // some code that performs file I/O handling
} catch (FileNotFoundException e) {
    // ....
} catch (EOFException e) {
    // ....
} catch (InterruptedException e) {
    // ....
}
```

Retrieving Information from the Exception Object

Previous error handling mechanisms were based on integer return codes that were limited in terms of the information they would carry. Java exceptions are true objects that can carry a boundless amount of information.

Typically, when receiving an exception, you can determine what went wrong by just looking at the type of the exception. Then it is up to you to do something useful in a catch block. In addition, the exception object provides a set of methods that enables you to retrieve some additional information. We list here the two most useful methods:

- ❑ `printStackTrace()` prints the exception object and the stack trace.
- ❑ `getMessage()` returns the string that has been passed to the constructor of the exception object.

Information



Java keeps track of all called methods and their context during the execution of a program. As Java keeps track of these methods and their context, it creates a list called the *call stack* or *execution stack*.

When an exception is thrown, each method in the call stack is checked to see whether it contains a catch block that matches (or is a superclass) of the exception thrown. If a catch block is found, program execution and the thrown exception object are passed to that block. If a matching catch block is not found, the program typically exits with an error.

The *stack trace* contains a list of all called methods.

Propagating an Exception

It is common to pass an exception up the execution stack until something can be done about the exceptional condition. In the following example the `InsufficientFundsException` is passed on, or *propagated*, up to the `goShopping` method:

```
void goShopping(BankAccount account, String shoppingList)
{
    try {
        getMoneyFromBank(account, 100.00);
    }
    catch(InsufficientFundsException e){
        // do something else
    }
}

void getMoneyFromBank(BankAccount account, double amount) throws
    InsufficientFundsException
{
    account.debit(amount);
}
```

In the above code snippet, the `getMoneyFromBank` method does not catch the `InsufficientFundsException` by declaring a try-catch block. Thus, the exception is passed on to the `goShopping` method.

You can also rethrow the exception, using the *throw* statement on the exception handle you receive from the catch block:

```
void goShopping(BankAccount account, String shoppingList)
{
    try {
        getMoneyFromBank(account, 100.00);
    }
    catch(InsufficientFundsException e){
        throw e;
    }
}
```

Notice now that the `goShopping` method rethrows the `InsufficientFundsException` object to exception handlers further up the call stack. In this case, all information about the exception object is preserved, and the handler at the higher context that catches the exception can extract the information from the object.

Java Standard Exceptions

Java provides a *Throwable* class that describes any object that can be thrown as an exception. Two general types of objects can be thrown: *Error*, which represents compile-time and system errors that you do not worry about catching, and *Exception*, which is thrown by the standard Java library class methods.

By throwing these exceptions in the standard Java library class methods, Java forces programmers to use the try-catch block structure in their code. In effect, a compilation error results from trying to call a method that throws an exception, outside a try block!

The Throwable Hierarchy

Figure 65 depicts the Throwable hierarchy that Java provides.

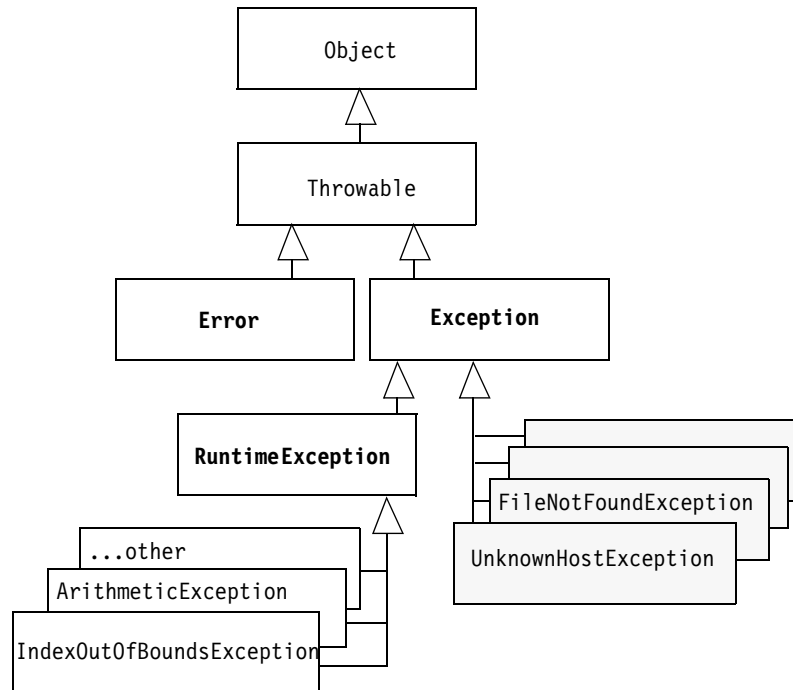


Figure 65. The Java Exception Hierarchy

Exception classes inherit from `Throwable`. You should only be concerned with the `Exception` subclasses as indicated by the shadowed boxes in Figure 65. The other classes belong to the run-time system.

There are three major groups of exception classes: *Error*, *Exception*, and *RuntimeException*.

Error

Error objects are for Java VM problems such as *InternalError* and *OutOfMemoryError*. Usually, you would not throw or catch these types of errors.

Exception

Exception objects are “expected” exceptions; that is, they are problems that have been foreseen. They are typically thrown when preconditions are violated: for example, if you try to open a non-existent file, a *FileNotFoundException* is raised.

RuntimeException

Run-time exceptions are typically programming errors, for example, an attempt to write beyond an array boundary, which would raise an *ArrayIndexOutOfBoundsException*. A *RuntimeException* may indicate a bug in your code.

Information



Any exception except subclasses of `RuntimeException` or `Error` that a method can throw are called *checked exceptions*. A method that can throw a checked exception must declare it, using the *throws* keyword in the method declaration.

The caller of the method must know which exceptions a method can throw and be able to react to those exceptions. If your code calls a method that throws an exception, your code must either put the method in a try block and catch the exception or declare that your code will throw that exception.

A `RuntimeException` or `Error` object does not have to be caught by any method, and it is not a documented part of a method's interface.

Using Predefined Exceptions

You use predefined exceptions by enclosing the methods that throw them in a try-catch block structure. For example, the following code catches a `FileNotFoundException` object and prints information to `System.out`:

```
try {
    java.io.FileInputStream in;
    // open a file for reading
    in = new java.io.FileInputStream("myfile");
}
catch (java.io.FileNotFoundException e) {
    System.out.println(e);
    e.printStackTrace();
    System.out.println(e.getMessage());
}
```

The output of the previous code snippet would look like this:

```
java.io.FileNotFoundException: myfile
java.io.FileNotFoundException: myfile
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
java.io.IOException(java.lang.String)
java.io.FileNotFoundException(java.lang.String)
java.io.FileInputStream(java.lang.String, boolean)
java.io.FileInputStream(java.lang.String)
evaluated code
myfile
```

Catching any Exception

When you write a catch block, you must specify the proper exception you want to catch. The exception type must match the exception thrown by the method you call in the try block. If several methods are called, you can treat the different exceptions they throw in separate catch blocks by cascading them as described in [Catching Multiple Exceptions](#) on page 144.

Matching an exception does not necessarily require a perfect match between the exception and a corresponding catch block. For example, you can create only one catch block that can catch any exception thrown by catching the base-class exception type *Exception*:

```
catch(Exception e) {
    System.out.println("Caught an exception");
}
```

Creating and Throwing Your Own Exceptions

Fortunately, you are not stuck with the already predefined exceptions that Java provides. You can create your own exception classes.

You create new instances of exception classes in the same way you create any other object: by using the *new* keyword. Then the new exception is thrown with the *throw* keyword:

```
public void debit(double anAmount) {
    if (getAvailableFunds() > anAmount){
        setBalance(getBalance() - anAmount);
        addTransaction(new Transaction(new Date(), -anAmount));
    }
    else{
        throw new InsufficientFundsException("Insufficient funds: "
            + getAvailableFunds());
    }
}
```

Defining a New Exception Class

You define a new exception class by extending an exception class already defined, preferably one that is close in meaning to your new exception. If you cannot find any suitable exception class, you can always derive your class from the `Exception` base class:

```
public class InsufficientFundsException extends Exception {
}
```

Constructors of Exception Classes

You instantiate or create a new exception just as you would create any other object: you use the *new* keyword and the constructor of the class. The existing constructors for the `Exception` class are:

```
public Exception()
public Exception(String s)
```

Your own exception constructor can call these constructors by using the *super* notation described in Method Overriding on page 107.

If you need more sophisticated constructors for passing arguments, you have to define your own constructors.

Implementing the InsufficientFundsException

In this section you add the `InsufficientFundsException` class to the existing `BankAccount` example. This exception is thrown by the `debit` method if the account does not have sufficient funds. You will:

1. Define a new exception class, `InsufficientFundsException`
2. Modify the `debit` method
3. Test the modified `BankAccount` class

First, copy the existing classes from the `ch06` package to a new package, `ch07`. Select the `ch06` package in the Workbench, select **Selected**→**Copy** and leave *Learn Java* as the entry in the Copy To field. Click **Finish** and enter `ch07` as the new package name.

Defining the InsufficientFundsException Class

To create your new exception class, follow these instructions:

1. Create a new class, *InsufficientFundsException*, in the `ch07` package. Change the Superclass entry field to *Exception* and click **Next>**.
2. Make sure the **Copy constructors from superclass** checkbox is selected and click **Finish**.
3. Change the constructor that takes a `String` to add the “Insufficient Funds Exception” string:

```
public InsufficientFundsException(String s) {  
    super( "Insufficient Funds Exception\n" + s);  
}
```

The constructor only adds “Insufficient Funds Exception” to the message string and passes the concatenated message string to the constructor of the superclass (see Source pane of Figure 66).

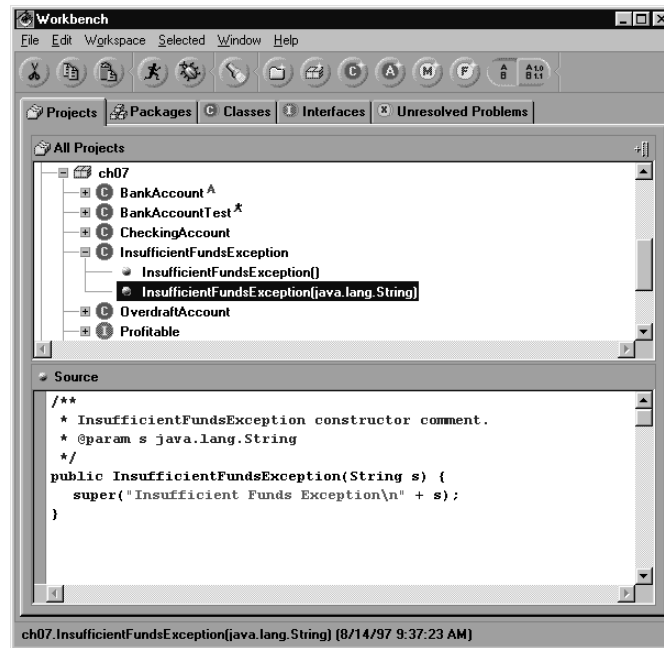


Figure 66. InsufficientFundsException Constructor

Modifying the Debit Method

To take advantage of the new exception you just created, you have to modify the `debit(java.util.Date,double)` method in the `ch07.BankAccount` class so that it throws the exception when the funds available are less than the requested amount:

```
public void debit(Date aDate, double anAmount)
    throws InsufficientFundsException
{
    if (anAmount > getAvailableFunds()){
        throw new InsufficientFundsException(
            "Insufficient available funds: " +
            getAvailableFunds());
    }
    else {
        setBalance(getBalance() - anAmount);
        addTransaction(new Transaction(aDate, -anAmount));
    }
}
```

Do not forget to change the definition of the *debit(double anAmount)* method so that it notifies method callers that it throws `InsufficientFundsException`:

```
public void debit( Date aDate, double anAmount)
    throws  InsufficientFundsException
```

Notice that you do not have to change the implementation of the *debit(double anAmount)* method because it reuses the code in the *debit(Date aDate, double anAmount)* method.

Testing the `BankAccount` Class

Use the `BankAccountTest` class to test the new exception. The class should currently show errors because you are calling the `debit` method without catching `InsufficientFundsException`. Change the main method in the `BankAccountTest` class to this:

```
public static void main( String argv[]) {
    CheckingAccount acct =
        new CheckingAccount("1234-5678");
    System.out.println("Exception Example\n");
    try {
        acct.credit( 30.0);
        acct.debit( 140.0);
    }
    catch ( InsufficientFundsException e) {
        System.out.println("Exception: " + e.getMessage());
    }
}
```

The try block contains the statements for adding \$30 and withdrawing \$140. You specify the possible exception in the catch block and simply format an error message that you print to the Console. Execute the code to see the error message in the Console window (Figure 67).

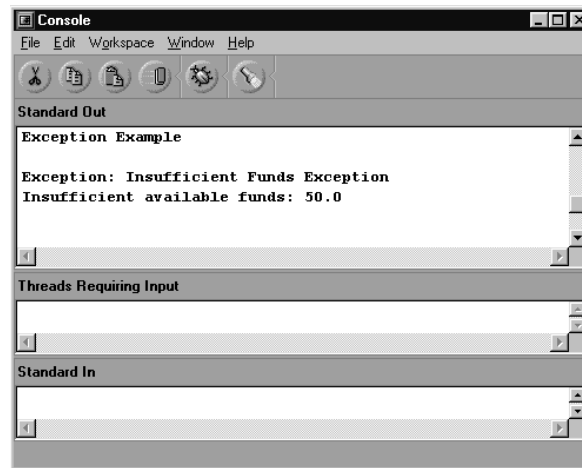


Figure 67. Running the Exception Example

For extra practice, you may want to improve the code in the catch block by attempting to withdraw only the amount available in the account.

Debugging a VisualAge for Java Program

Now that you know how to handle run-time errors in your programs, you can learn more about analyzing and fixing errors during development.

VisualAge for Java provides several tools to find problems in your programs. They include the Scrapbook, which you know already, the **Inspectors**, and the **Debugger**.

In this section, we introduce you to the Inspectors and the Debugger, your primary weapons for eradicating bugs from your code. Because knowing Inspectors is a prerequisite for using the Debugger efficiently, we introduce Inspectors first.

Inspectors

Using Inspectors, you can view and change the state of objects in your programs. If you are working in the Scrapbook or the debugger, follow these steps to open an Inspector:

1. Open the Scrapbook and type the following code:

```
new java.awt.Point(1,2);
```

2. Highlight the code and from the menu bar choose **Edit→Inspect**. An inspector window similar to Figure 68 is opened.



Figure 68. An Inspector Window

Tip



An alternative way to open an Inspector is to insert an instruction in your Java code to inspect a specific object. This is a useful method if you have a difficult-to-debug program that should not be interrupted with breakpoints. In the Scrapbook type the following code:

```
String[] numbers = {"one", "two", "three"};
uvm.tools.DebugSupport.inspect(numbers);
```

Highlight the code and from the menu bar choose **Edit→Run**. The `uvm.tools.DebugSupport.inspect(anObject)` instruction advises the Java VM to open an Inspector on the specified object.

The Inspector Window

The title bar of an Inspector window displays the type of the displayed class and the context in which you have opened the Inspector. In the example in Figure 68, the class is a *java.awt.Point*, and the context is *Page 1 of the Scrapbook*. Two panes are part of in the Inspector window:

Fields pane Shows the fields of the object

Value pane Displays the values of the fields of an object

The Fields pane displays items in hierarchical order with the inherited fields first. Therefore, when you inspect a more complex object, the fields that you have declared in your object are at the bottom of the Fields pane.

Changing the Value of a Field

In the Value pane of the Inspector window you can manipulate the values of the object's fields (Figure 69). Select the *int x* field and change *1* to *100*, then choose **Edit**→**Save** from the menu bar. Alternatively you can access the Save option by using the pop-up menu of the value pane.

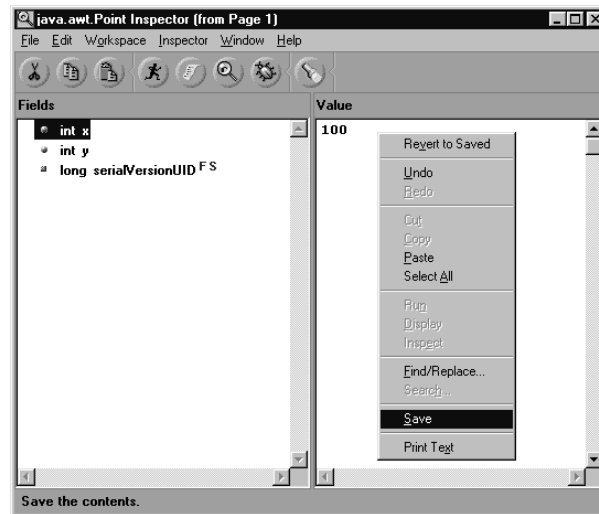


Figure 69. Changing the Value of a Field

When stepping through your code with the Debugger, you can change a field value on the fly and resume the execution of your program.

Navigating to Other Fields or Objects

If you have a more complex object, say, an aggregation of several objects, it is easy to open an Inspector on the other objects. Just select the other object in the Fields pane and choose **Inspector**→**Inspect** from the menu bar. You can in this way inspect very complex data structures by accessing their different layers, as you would peel an onion.

Read This

Because the String class is designed for immutable strings, the Inspector window does not allow you to change the characters of a String object. However, you can change the characters of a StringBuffer. In this case, you must specify the new character enclosed in single quotes.

Evaluating Code in the Context of an Object

If you open an Inspector on a particular object you can apply methods to that object and inspect the results. In Figure 70, for example, the *getLocation()* method is invoked on a *Point* object.

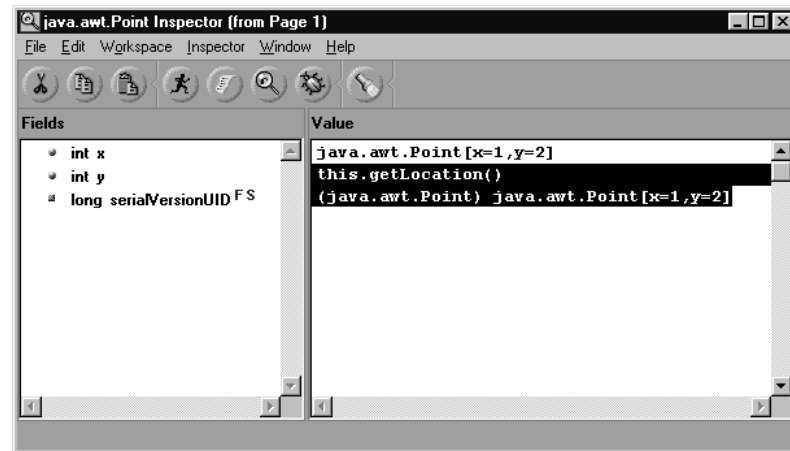


Figure 70. Evaluating Code in the Context of an Object

To evaluate an expression, type in the expression in the Value pane, highlight it, and choose **Display** from the menu bar or from the Value pane pop-up menu.

Hiding Nonpublic Fields

When an Inspector first appears, all public, protected, and private fields of the inspected class and the inherited classes are displayed. To limit the display to public fields, choose **Inspector**→**Public Fields** from the menu bar.

The Debugger

The Debugger allows you to step through your Java code, fix the code, and inspect and change the state of objects.

In this section you learn how to:

- ☐ Insert and remove breakpoints
- ☐ Start a debug session
- ☐ Step through the methods in your programs
- ☐ Debug and fix a program

For our example we use the `BankAccount` class.

Inserting and Removing Breakpoints

The Debugger starts automatically if a program running in the VisualAge for Java environment reaches a breakpoint or throws an uncaught exception. In this section you set a breakpoint in the `debit` method of the `BankAccount` class.

To set a breakpoint, perform these steps:

1. Select the `ch07.BankAccount.debit(java.util.Date, double)` method from the Workbench or from the class browser.
2. In the Source pane of the `debit` method, set the cursor to the first line of code: `if( anAmount > getAvailableFunds() ) {`.
3. From the menu bar select **Edit→Insert/Remove Breakpoint**. A breakpoint icon appears to the left of the statement as shown in Figure 71. (To remove the breakpoint, simply select **Edit→Insert/Remove Breakpoint** again.) Alternatively, you can access the pop-up menu of the Editor or double-click in the left margin of the line of code to set or remove a breakpoint.

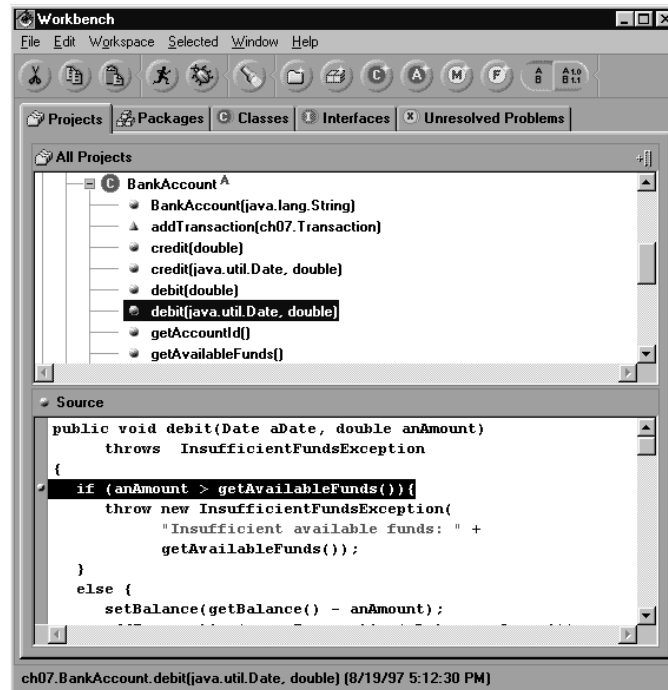


Figure 71. Breakpoint Set in the Debit Method

Now you are ready to start the debug session and step through the methods.

Tip



You can access the list of all breakpoints defined in your current workspace by selecting **Window→Breakpoints** from the Workbench menu bar. From the Breakpoint window, you can:

- Remove all breakpoints
- Remove selected breakpoints
- Enable breakpoints
- Disable breakpoints

Starting a Debug Session

Use the Scrapbook to start a debug session. Execute the following code in the Scrapbook (do not close the Scrapbook, you will need it in a minute):

```
ch07.CheckingAccount acct = new ch07.CheckingAccount("1234-5678");
try {
    acct.credit(30.0);
}
```

```

        acct.debit(140.0);
        System.out.println("new balance: " + acct.getBalance());
    }
    catch (ch07.InsufficientFundsException e)
    {
        System.out.println("Exception: " + e.getMessage());
    }
}

```

Program execution stops at the previously defined breakpoint, and the Debugger window opens as shown in Figure 72.

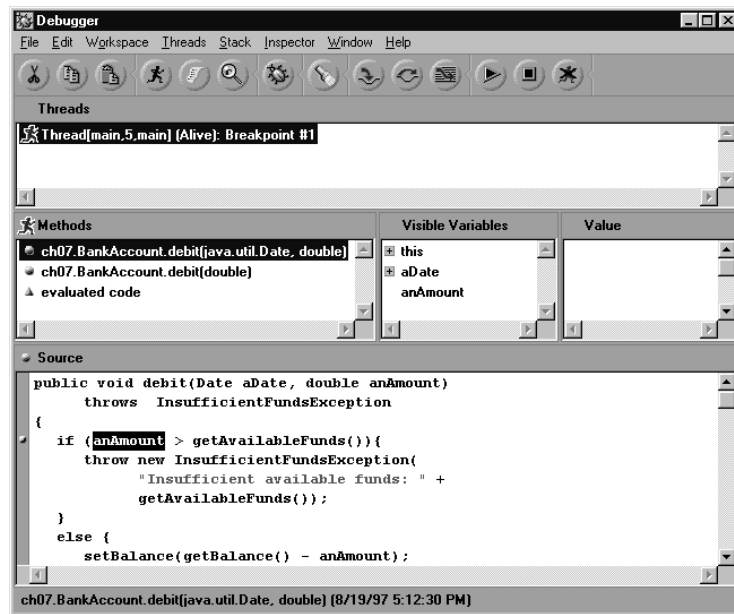


Figure 72. Debugger Window

The VisualAge for Java Debugger window is divided into five panes:

Threads

The Threads pane displays the thread in which your program is running.

Methods

In the Methods pane, each line represents a stack frame, which corresponds to a method that was called. In our example the highlighted code and the *debit* method represent a stack frame. The top item in the pane is the current stack frame.

Visible Variables	The Visible Variables pane displays locally visible variables. You can view the current object in more detail by expanding the tree of the object. The current object is represented by the <i>this</i> keyword, and you expand the tree by clicking +.
Value	The Value pane displays the value of a selected field in the variable pane.
Source	The Source pane contains the code of the method through which you are stepping.

You can maximize each pane by double-clicking its title. To restore the original pane size, double-click one more time on the pane title.

Stepping through the Methods

The navigation buttons on the debugger toolbar enable you to control the way the debugger steps through your methods. The navigation buttons are located at the right end of the toolbar (Figure 73).

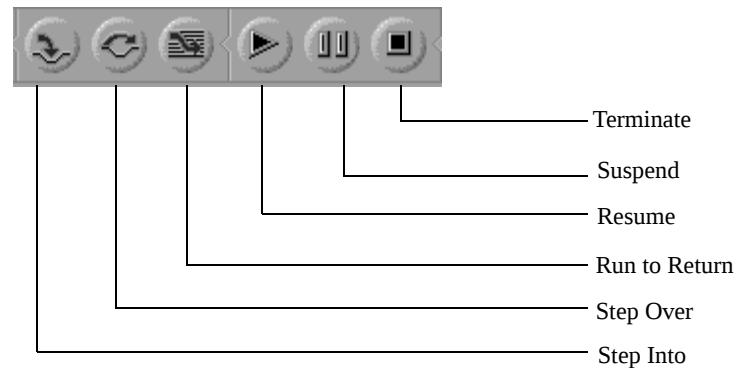


Figure 73. Debugger Navigation Toolbar Buttons

If an executing program hits a breakpoint, the Debugger opens and the navigation buttons are enabled. Use these buttons to navigate through your code:

Step Into Steps into the next statement. If the statement invokes a method, the Source pane of the Debugger window displays the source of the method into which you stepped.

Step Over	Executes the statement that is currently selected in the Debugger window. The values of the variables and fields, displayed in the Text pane, are updated.
Run to Return	Continues execution to the end of the current method and stops.
Resume	Resumes normal program execution. The current thread where your program runs will be removed from the VisualAge for Java Debugger window Threads pane.
Suspend	Suspends program execution.
Terminate	Terminates program execution.

To become familiar with the navigation buttons, step over the statements until you reach the end of the program. An exception is thrown and a message is printed to the Console.

Debugging and Fixing the Program

An exception is thrown by the *debit* method because the available funds are insufficient. To avoid this, change the value of the *balance* field to a larger value during your debug session. Follow these steps:

1. Restart the Debugger by running the code in the Scrapbook again (Figure 74).
2. In the Visible Variables pane of the Debugger, expand the CheckingAccount object by choosing *this* and clicking +.
3. Select the *double balance* field. The balance is 50 dollars, (20 to open the account and a credit of 30).
4. In the *Value* pane change the value to **240** and save it by using the pop-up menu and choosing **Save** (Figure 74).
5. Step over the statements. The exception is not thrown in this debug session because you changed the *balance* to a sufficient value. Finish the debug session by stepping to the end or by clicking the **Resume** button.

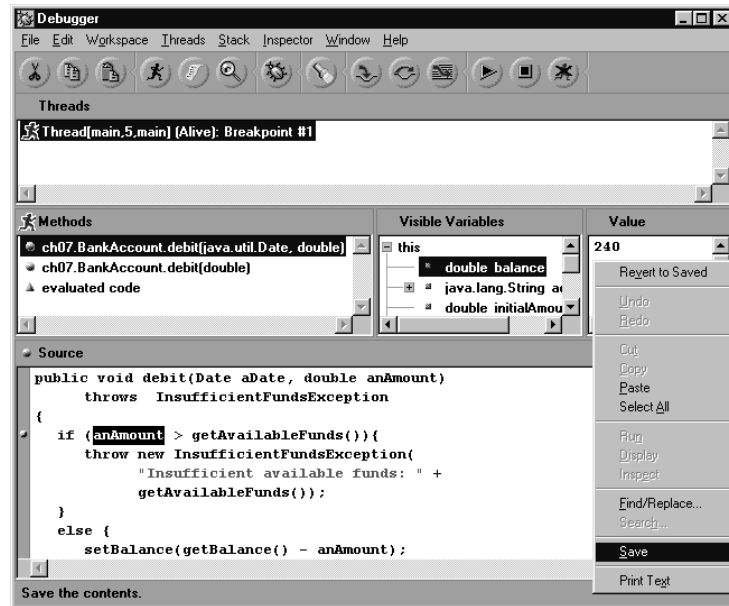


Figure 74. Changing a Value During a Debug Session

Summary



Java uses the try-catch-finally block structure to catch and handle exceptions that are thrown by methods. Java exceptions are classified into three categories: Error, Exception, and RuntimeException.

Subclasses of Exception are checked exceptions; that is, they are a part of the interface of a method, and the caller must catch or throw the defined exceptions.

VisualAge for Java provides Inspectors and an integrated Debugger for finding bugs and stepping through the code in your programs.

8

File I/O and Persistence

Up to now, each time you run your samples they create new bank accounts from memory. Of course, in a real bank the accounts objects would probably be stored in files or databases, and you would not have to create new accounts each time you want to make new transactions. After all, customers might become angry if their account balance is reset to a minimum amount of \$20 each time they access their account to make a withdrawal!

JDK 1.1 introduces a very useful feature called *object serialization* that enables you to convert an object to a sequence of bytes that can later be restored to the original object and thus provides object persistence.

Bringing persistence to your account objects is exactly what you do in this chapter. After a brief introduction to the input/output (I/O) library supported in Java to communicate with files, console, or network connections, you learn how to store and read your objects to and from files, using the *Serializable* interface.

Persistence

Most programs need some sort of persistence, ranging from a simple initialization file for user preferences to a large database that stores records for a utility company. Java supports these requirements, for example, through the *Properties* class and the JDBC interface, as well as providing thorough support for reading and writing files. In this chapter, we focus on reading and writing objects from and to files. For more information about other I/O capabilities of Java, refer to the JDK 1.1 documentation.

In this chapter, we want to save, or make *persistent*, our bank account information. In many cases, persistence needs can be satisfied by simply writing ASCII text to a file. For example, the bank account could write out the account number and balance as two strings. However, it would be helpful if you could read actual objects from files almost transparently rather than reading strings and then building objects from them by using a conversion code.

Reading and writing objects is not a trivial task. Objects must be *serialized* or *flattened* before they can be written to a file. That is, all the information the object contains must be converted to a sequence of bytes that can be written to a text file or transferred across the network. When the object is read back into a program, the information must be restored or *resurrected* to the original object.

The reading and writing of objects in Java is supported through the *Serializable* interface.

Java I/O

In this section, we provide a short introduction to Java I/O to help you understand the serialization process.

In Java, I/O is handled through *streams*. A stream is a path of communication between a source of information and a destination. Using streams, your programs can transmit a series of bytes or characters from a source to a destination, which could be as various as files, machines on a network or the keyboard (standard input), and the console (standard output) of your computer.

In Java, the I/O class library is stored in a package called *java.io*. The *java.io* package can generally be divided into three groups of classes providing I/O for *input*, *output*, and *object* streams.

In this chapter we focus on the object stream classes. For more information about the input or output stream classes, consult the JDK 1.1 documentation.

Input Streams

Input streams enable you to receive data from a stream. Java provides two types of classes to read streams: 8-bit byte stream classes, which have a suffix of *Stream*, and 16-bit character stream classes, which have a suffix of *Reader*. The 16-bit stream classes are suitable for internationalization because they support the 16-bit Unicode Java's native char format.

Java provides abstract classes, *InputStream* and *Reader*, that define the protocol for all input streams. By inheritance, all classes that derive from either one of these abstract classes have basic methods called *read* for reading a stream.

Input stream classes can be divided into simple input stream classes or filtered input stream classes depending on the type of stream they manipulate.

Simple Input Streams

The *InputStream* and *Reader* abstract classes and their derived classes provide an interface to read raw data from a file. They are usually used in conjunction with the other stream classes. The nonabstract classes that derive from *InputStream* and *Reader* can be listed according to their input stream:

- ❑ *FileInputStream* and *FileReader* read information from a file.
- ❑ *PipedInputStream* and *PipedReader* read information from a pipe, which channels output written, respectively, by a *PipedOutputStream* or a *PipedWriter*.
- ❑ *ByteArrayInputStream* and *CharArrayReader* read from a buffer in memory.
- ❑ *SequenceInputStream* concatenates several *InputStream* objects into a single *InputStream*.
- ❑ *StringBufferInputStream* and *StringReader* read from a *StringBuffer* object.

Filtered Input Streams

FilterInputStream and *FilterReader* are abstract classes derived respectively from *InputStream* and *Reader*. These classes and their subclasses define the interface for filtered streams, which

process data as it is being read. They enhance input stream function and can be used depending on the way input streams behave internally:

- ❑ `BufferedInputStream` and `BufferedReader` provide improved read operations by buffering data while reading the input stream. These classes are useful when there are large amounts of data or fast access is required.
- ❑ `DataInputStream` reads primitive Java data types in a machine-independent format.
- ❑ `LineNumberInputStream` and `LineNumberReader` keep track of line numbers while reading.
- ❑ `PushbackInputStream` and `PushbackReader` read an input stream, using a one-byte pushback buffer. These classes can be useful when you are reading data from a stream and you need to peek at the next character in the stream to decide what to do next. If you peek at a character in an ordinary stream, you need to put it back so that it can be read again and processed normally.

Output Streams

Output streams enable you to send data as a stream. To mirror the `InputStream` and `Reader` abstract classes, Java provides the `OutputStream` and `Writer` abstract classes, which define the protocol for all output streams. By inheritance, all classes that derive from either one of these abstract classes have basic methods called *write* for writing to a stream.

Output stream classes can be divided into simple output streams classes or filtered output stream classes depending on the type of stream they manipulate.

Simple Output Streams

The `OutputStream` and `Writer` abstract classes and their derived classes provide an interface to write raw data to a file. They are usually used in conjunction with the other stream classes. The nonabstract classes that derive from `OutputStream` and `Writer` can be listed according to their output stream:

- ❑ `FileOutputStream` and `FileWriter` write information to a file.
- ❑ `PipedOutputStream` and `PipedWriter` write information to a pipe which can be read, respectively, by a `PipedInputStream` or a `PipedReader`.
- ❑ `ByteArrayOutputStream` and `CharArrayWriter` write to a buffer in memory.

- ❑ `StringWriter` reads from a `StringBuffer` object.

Filtered Output Streams

`FilterOutputStream` and `FilterWriter` are abstract classes derived, respectively, from `OutputStream` and `Writer`. These classes and their subclasses define the interface for filtered streams, which process data as it is being written. They enhance output stream function and can be used depending on the way output streams behaves internally:

- ❑ `BufferedOutputStream` and `BufferedWriter` provide improved write operations by buffering data while writing to the output stream. These classes are useful when there are large amounts of data or fast access is required.
- ❑ `DataOutputStream` reads primitive Java data types in a machine-independent format.
- ❑ `PrintStream` and `PrintWriter` produce formatted output and handle the display of data by comparison to `DataOutputStream` which handles the storage of data.

Utility Classes

Three additional classes are provided in the `java.io` package to let you manipulate files and input streams more efficiently: the `File` class, the `RandomAccessFile` class and the `StreamTokenizer` class.

File Class

The `File` class is not a surrogate of a computer file. Rather, it provides you with useful methods to manipulate file names for a particular file or a set of files in a directory.

RandomAccessFile Class

The `RandomAccessFile` class is used for files containing records of a known size and allows you to randomly access these records using the `seek()` method.

StreamTokenizer Class

The `StreamTokenizer` class can be used only in association with an `InputStream` object to break it into a sequence of tokens delimited by whatever you choose. This class is obviously very useful for parsing an input stream.

Using Input/Output Streams

To conclude this quick overview of the Java I/O classes, we present a simple program that demonstrates how you can use input and output streams to read bank account IDs from standard input, write them in a file, and read the file to regenerate the bank account objects:

```
System.out.println("=== Reading from the console ===");
java.io.DataInputStream in = new java.io.DataInputStream(
    new java.io.BufferedInputStream(System.in));
String s;
java.util.Vector accountList = new java.util.Vector();

try {
    while ((s = in.readLine()).length() != 0) {
        ch07.CheckingAccount ch = new ch07.CheckingAccount(s);
        System.out.println(ch);
        accountList.addElement(ch);
    }
} catch (java.io.IOException e) {
    e.printStackTrace();
}

System.out.println("=== Writing File: DATA.TXT ===");
java.io.DataOutputStream outFile = new java.io.DataOutputStream(
    new java.io.BufferedOutputStream(
        new java.io.FileOutputStream("Data.txt")));

try {
    for (int i = 0; i < accountList.size(); i++) {
        s = ((ch07.CheckingAccount)accountList.elementAt(i)).
            getAccountId();
        System.out.println("Writing: " + s);
        outFile.writeBytes(s + "\n");
    }
    outFile.close();
} catch (java.io.IOException e) {
    e.printStackTrace();
}

System.out.println("=== Reading File: DATA.TXT ===");
java.io.DataInputStream inFile = new java.io.DataInputStream(
    new java.io.BufferedInputStream(
        new java.io.FileInputStream("Data.txt")));

try {
    while ((s = inFile.readLine()) != null) {
        ch07.CheckingAccount ch = new ch07.CheckingAccount(s);
        System.out.println(ch);
    }
}
```

```
catch (java.io.IOException e) {  
    e.printStackTrace();  
}
```

In the first part of the program, account identifiers are read from the Console input, and `CheckingAccount` objects are allocated in a `Vector`. In the second part of the program, we loop through the `Vector` object and write the identifier of each account in the `DATA.TXT` file. Finally, we read back the file and dynamically re-create each account.

You can directly type this code in the Scrapbook and execute it. When running the code, the Console window is displayed. Enter the account identifier in the Standard In multiline editor and validate it by pressing Enter (Figure 75).

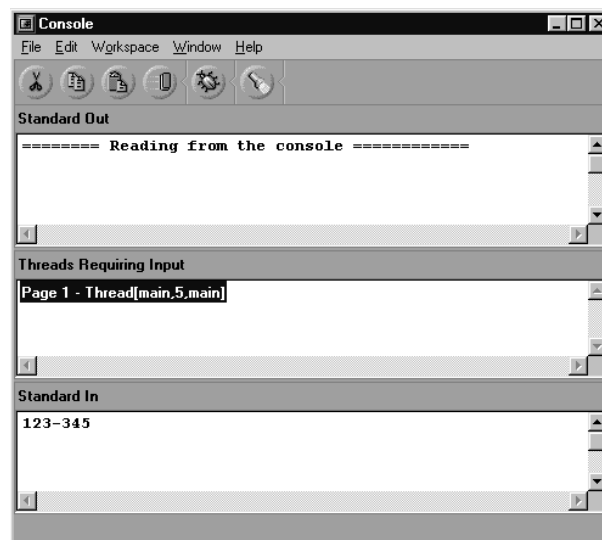
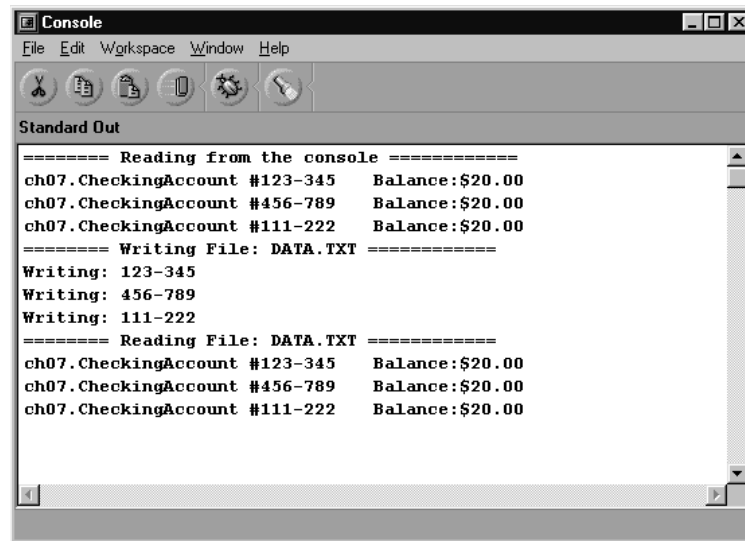


Figure 75. Reading Account Identifier from Standard Input

When you have entered enough accounts, press Enter one more time to write the file and read it back. The Console window should display your accounts read from the `DATA.TXT` file (Figure 76). You can verify that the file contains your entries by looking it up in `\IBMVJava\Ide\Program`.

A screenshot of a 'Console' window from the VisualAge for Java IDE. The window has a menu bar with 'File', 'Edit', 'Workspace', 'Window', and 'Help'. Below the menu is a toolbar with icons for file operations and execution. The main area is titled 'Standard Out' and contains the following text:

```
===== Reading from the console =====
ch07.CheckingAccount #123-345    Balance:$20.00
ch07.CheckingAccount #456-789    Balance:$20.00
ch07.CheckingAccount #111-222    Balance:$20.00
===== Writing File: DATA.TXT =====
Writing: 123-345
Writing: 456-789
Writing: 111-222
===== Reading File: DATA.TXT =====
ch07.CheckingAccount #123-345    Balance:$20.00
ch07.CheckingAccount #456-789    Balance:$20.00
ch07.CheckingAccount #111-222    Balance:$20.00
```

Figure 76. Output from Program

Object Streams

Object streams are high-level streams that enable you to read and write objects on file or network streams. In the rest of this chapter we focus on creating object streams for the bank account example as an alternative to using I/O streams.

Object Serialization

JDK 1.1 provides you with a new feature called *object serialization*. It allows you to transform any object that implements the `Serializable` interface into a byte stream that can be used later to fully restore the original object. This feature is designed to work not only on a single machine but also across a network. Thus, you can *flatten* an object on one machine and send the byte stream representation of the object over the network to another machine, which can then *resurrect* the object to use it. The object serialization mechanism and the Java VM take care of any differences between operating systems or data representation on different machines: Everything is automatic. Well, almost!

The Serializable Interface

The Serializable interface is part of the *java.io* package. It provides lightweight persistence for your objects so that they can live beyond the existence of a running program. When your object is serializable, it can be written to disk and restored later on when the program is reinvoked.

Information



In contrast to object-oriented databases, which provide you with transparent persistence for your objects, in Java you still have to call in your code specific methods to read or write your objects from or to the disk. That is why we say that the serialization mechanism provides only *lightweight* and not full persistence.

To make your object serializable:

1. Implement the Serializable interface in all of your classes for which you want the object to be persistent.
2. Mark any computed or redundant fields as *transient*. These fields will not be saved when you save the object.
3. To serialize an object, create some sort of OutputStream and wrap it inside an ObjectOutputStream object. Then use the writeObject method to serialize the object and send it to the OutputStream.
4. To deserialize an object, wrap an InputStream object inside an ObjectInputStream object and call the readObject to read the object from the InputStream.

Figure 77 shows the serialization process.

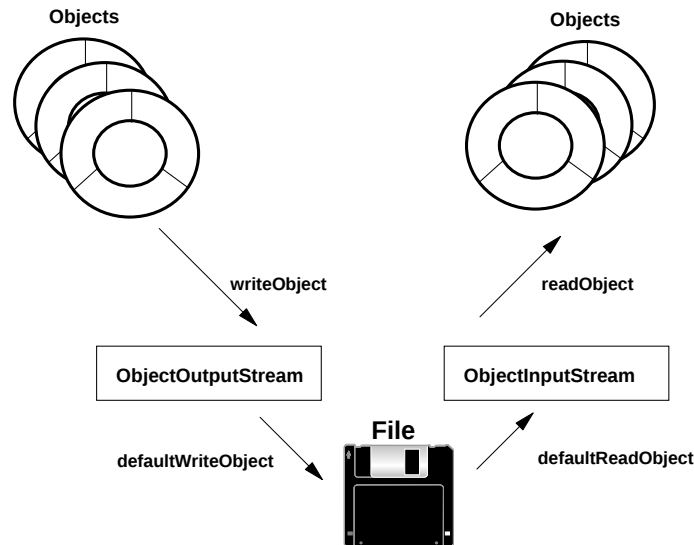


Figure 77. The Serialization Process

The process of serialization is achieved through the use of `ObjectOutputStream` and its `writeObject` method. The objects are written on the `ObjectOutputStream` (by the `defaultWriteObject` method) and flattened to the file (including the fields, type information, and references to other objects).

The restoration of the object is achieved through the use of `ObjectInputStream` and its `readObject` method. The `ObjectInputStream` reads the data from the file and restores the objects (type, data, and dependencies). This process is handled automatically by the `defaultReadObject` method.

Object Dependencies

A clever aspect of the serialization mechanism is that not only the object but all its dependencies are serialized or deserialized during `writeObject` or `readObject` execution. Therefore if your object is built by *aggregation* and contains the references or handles of other object, those objects also will be serialized as long as their classes implements the `Serializable` interface and they have not been flagged with the *transient* keyword.

Behind the Scene of Serialization

As instances of a class implementing the `Serializable` interface, objects have default behavior supplied by Java for writing their data to an output stream. The default behavior is to write the following information:

- ☐ Object's class
- ☐ Class signature
- ☐ Values of the fields of its superclass
- ☐ Values of nontransient and nonstatic fields

When primitive types are serialized, a simple call to the corresponding `OutputStream` write method is made. For objects, the serialization process is done through a call to the *writeObject(anObject)* method of *ObjectOutputStream*. This method checks whether the instance is an instance of `Serializable` and has an overridden `writeObject` method. If so, it uses the overridden method; otherwise it uses the *defaultWriteObject* method defined in *ObjectOutputStream*.

When an object is read and deserialized, the class type, class signature, values of its superclass's nontransient and nonstatic fields, and class fields are read. Objects referenced by this object are read transitively so that a complete equivalent graph of objects is reconstructed by `readObject`. `readObject` then returns an object of type `Object`. You need to cast it back to the original class to use it.

The default deserialization for a class is handled by the default-`readObject` method of `ObjectInputStream` and can be modified by overriding the `readObject` of the class (this must be done in conjunction with the override of `writeObject`).

If you have marked some fields as transient, you should override the `readObject` method to re-create these redundant fields after the default `readObject` method has restored the nontransient fields.

Controlling the Serialization

To gain more control during the serialization or deserialization of an object, it is possible to provide customized read and write methods that are responsible for reading and writing the object's state. In this case the stream is only responsible for storing the name of the object's class.

The Externalizable interface identifies objects that can be saved to a stream but are responsible for saving their own state. For this purpose, externalizable objects must provide a `writeExternal` method for storing their state during serialization and a `readExternal` method for restoring their state during deserialization.

BankAccount Serialization

In this section you make the BankAccount objects persistent by implementing and testing the Serializable interface first for the Transaction class and then for the BankAccount class.

Serializing the Transaction Class

In this section you:

- ☐ Modify the Transaction class to implement the Serializable interface
- ☐ Test the serialization mechanism

You have to copy your previous classes into the new package for the chapter. Select the `ch07` package in the Workbench, select **Selected**→**Copy**, and click **Finish**. Enter `ch08` in the Copy dialog that appears and click **OK**.

Implementing the Serializable Interface

Select the **ch08.Transaction** class in the Workbench and modify the class definition to reflect the following changes (shown in bold):

```
import java.util.*;
import java.io.*;
public class Transaction implements Serializable {
    private Date date = null;
    private double = 0.0;
}
```

It is that simple; your class can now be made persistent!

Testing the Serialization Mechanism for the Transaction

To test your serializable Transaction class, follow these instructions:

1. Create a new class called *TransactionTest* in the *ch08* package, which is derived from the *Object* class. Select the **main(String[])** checkbox on the Attributes page of the SmartGuide. Complete the class definition and main method to match the code shown in Step 2. Notice that you need to add the import statements to the class declaration and the body of the method to the main method.
2. Change the value of *fileName* (currently *c:\tmp\trans.ser*) to a directory that exists in your environment. Notice the use of the *file.separator* property. This allows you to run the code on platforms that use different path separators (such as Windows NT and UNIX).

```
import java.util.*;
import java.io.*;

public class TransactionTest{
    public static void main( String[] args) {

        String sep = System.getProperty("file.separator");
        String fileName = "c:"+sep+"tmp"+sep + "trans.ser";
        Transaction trans1 =
            new Transaction(new Date(), 1000.00);
        Transaction trans2 =
            new Transaction(new Date(), -2000.00);
        FileOutputStream fOut = null;
        ObjectOutputStream out = null;
        System.out.println("Serializing the objects.");
        try {
            fOut = // use an appropriate path (c:\tmp)
                new FileOutputStream( fileName);
            out = new ObjectOutputStream( fOut);
            out.writeObject( trans1);
            out.writeObject( trans2);
            out.close();
            fOut.close();
        }
        catch( IOException ioExc) {
            ioExc.printStackTrace();
        }
        System.out.println("Done!");
    }
}
```

3. Save your work and test it by selecting the *TransactionTest* class and then **Selected**→**Run**→**Run main...** or by clicking the **Run** tool bar button.

Check that the application has created a file called `trans.ser` in the directory you specified.

To read the file and restore the serialized objects, append the following code to the end of the main method:

```
try {  
  
    trans1 = trans2 = null;  
    System.out.println("Reading the variables back in.");  
  
    FileInputStream fIn = new FileInputStream( fileName);  
    ObjectInputStream in = new ObjectInputStream( fIn);  
  
    trans1 = (Transaction)in.readObject();  
    trans2 = (Transaction)in.readObject();  
    System.out.println( trans1 + "\n" + trans2);  
    in.close();  
    fIn.close();  
}  
catch( OptionalDataException ioExc) {  
    System.out.println(ioExc);  
}  
catch( ClassNotFoundException exc) {  
    System.out.println(exc);  
}  
  
catch( IOException ioExc) {  
    System.out.println(ioExc);  
}  
System.out.println("Done!");
```

In this new piece of code, you create a `FileInputStream` object from the `.ser` file and pass it to the `ObjectInputStream` constructor. Then you restore the two objects from the file and display them in the Console window as in Figure 78.

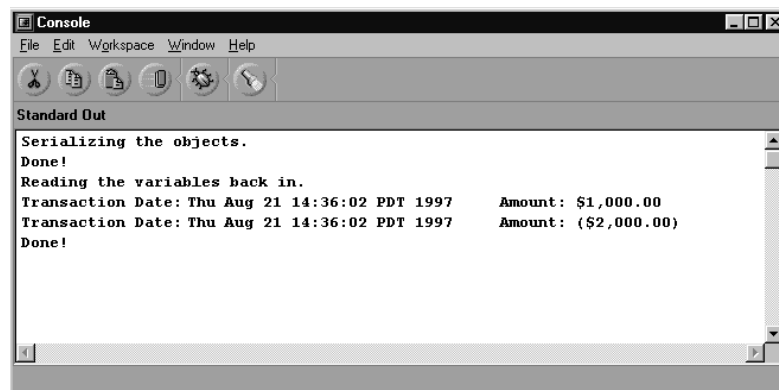


Figure 78. TransactionTest Output

Serializing the BankAccount Class

In this section you:

1. Implement the Serializable interface in the BankAccount class
2. Modify the BankAccountTest class to test the serialization mechanism.

Implementing the Serializable Interface

Modify the class definition of ch08.BankAccount as shown in bold:

```
import java.io.*;
public abstract class BankAccount implements Serializable
```

Modifying the BankAccountTest Class

To modify your BankAccountTest class and test the latest changes to BankAccount, follow these instructions:

1. Add the import java.io.* statement in the class declaration of the BankAccountTest class.
2. Change the code of the main method to the following:

```
public static void main( String[] args) {
    String sep = System.getProperty("file.separator");
    String fileName = "c:" + sep + "tmp" + sep + "account.ser";
    FileOutputStream fOut = null;
    ObjectOutputStream out = null;
    CheckingAccount accnt =
        new CheckingAccount("1234-5678");
    System.out.print("Writing Persistent Account Example\n");
    try {
        accnt.credit( 10000.00);
        accnt.debit( 1234.00);
        accnt.debit( 344.00);
        accnt.credit( 1000000.00);
    }
    catch ( InsufficientFundsException e) {
        System.out.println("Exception: " + e.getMessage());
    }
    System.out.println( "\n" + accnt.toString() + "\n");
    // Let's make the bankAccount persistent now.
    try {
        fOut = new FileOutputStream( fileName);
        out = new ObjectOutputStream( fOut);
        out.writeObject( "Flattened on " + new Date());
        out.writeObject( accnt);
    }
```

```
    }  
    catch( FileNotFoundException exc) {  
        exc.printStackTrace();  
    }  
    catch( IOException exc) {  
        exc.printStackTrace();  
    }  
    try {  
        out.close();  
        fOut.close();  
    }  
    catch( IOException exc) {  
        exc.printStackTrace();  
    }  
    System.out.print("Done\n");  
}
```

3. Execute the `BankAccountTest` class and verify that the `account.ser` file is created in the appropriate directory (`c:\tmp` by default). You can then read the objects back with the following code appended to the `main` method:

```
FileInputStream fIn = null;  
ObjectInputStream in = null;  
System.out.print("Reading Persistent Account Example\n");  
// Let's create the bankAccount from the persistent storage.  
try {  
    fIn = new FileInputStream(fileName);  
    in = new ObjectInputStream(fIn);  
    System.out.println((String)in.readObject());  
    acct = (CheckingAccount)in.readObject();  
    System.out.println( "\n" + acct.toString() + "\n");  
}  
catch(ClassNotFoundException exc) {  
    exc.printStackTrace();  
}  
catch(FileNotFoundException exc) {  
    exc.printStackTrace();  
}  
catch(IOException exc) {  
    exc.printStackTrace();  
}  
try {  
    in.close();  
    fIn.close();  
}  
catch(IOException exc) {  
    exc.printStackTrace();  
}  
System.out.print("Done\n");
```

When executing the `main` method, the Console window should display the `BankAccount` objects as shown in Figure 79.


```

Console
File Edit Workspace Window Help
Standard Out
Writing Persistent Account Example
ch08.CheckingAccount #1234-5678      Balance:$1,008,442.00
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: $10,000.00
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: ($1,234.00)
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: ($344.00)
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: $1,000,000.00

Done
Reading Persistent Account Example
Flattened on Thu Aug 21 14:19:41 PDT 1997
ch08.CheckingAccount #1234-5678      Balance:$1,008,442.00
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: $10,000.00
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: ($1,234.00)
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: ($344.00)
Transaction Date: Thu Aug 21 14:19:40 PDT 1997      Amount: $1,000,000.00

Done

```

Figure 79. Restoring the BankAccount Objects.

Summary



Java provides comprehensive I/O support including support for making objects persistent.

The `Serializable` interface provides automatic support for making your objects persistent.

If you have more complex objects or objects with transient properties, you have to create `readObject` and `writeObject` methods for your serializable objects to restore their original values.

9

Managing Your Code

VisualAge for Java comes with a powerful set of tools to store and manage updates in your programs. All activity in VisualAge for Java is organized around a single, persistent workspace, which contains the Java programs on which you are working. The code and its different editions that you produce from this workspace are automatically stored in the repository along with other projects and can be restored later during a new working session.

In this chapter we describe the workspace and the repository and explain how to work with multiple editions of program elements to facilitate code management. We also introduce you to the repository browser and the search engine that you can use to find workspace program elements, such as classes, interfaces, fields, or methods.

Storing Your Code

When you start VisualAge for Java, a persistent working environment, called a *workspace*, is loaded automatically to store your program elements (projects, packages, classes, interfaces, and methods).

During your working session, your code is also stored in a larger database, called the *repository*, which contains other projects and program element in a defined state, called *edition*, that you can add to the workspace if you need to use them.

Read This



VisualAge for Java has a wealth of examples that are not shown in your workspace the first time you start the product. These examples are stored in your repository and need to be loaded in your workspace. Follow the instructions described in Loading Available Editions on page 188 to do so.

As shown in Figure 80, Java code is *loaded* from the repository into the workspace and *saved* from the workspace to the repository. The Workbench and other browsers are used to manage code in the workspace; the Repository Explorer is used to browse code in the repository.

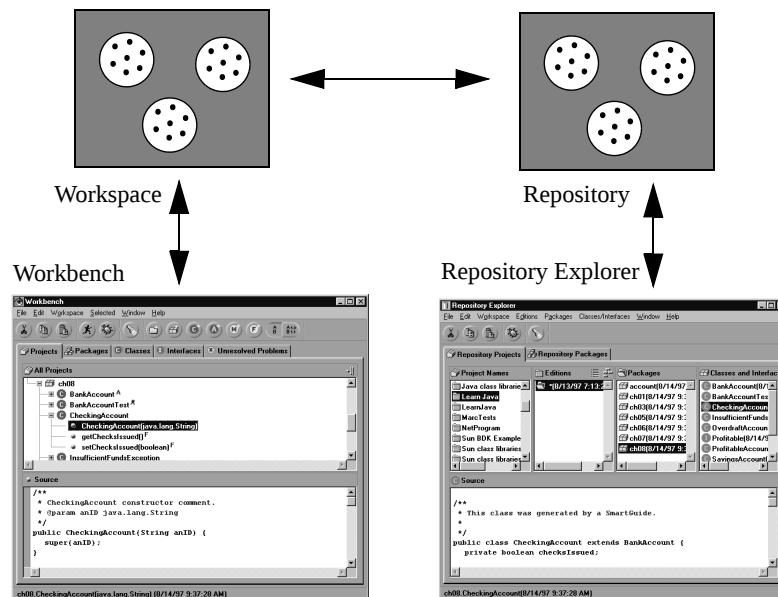


Figure 80. Managing your Code in VisualAge for Java

Read This

This book describes the single-user environment of the Professional version of VisualAge for Java. The Enterprise version of VisualAge for Java has a more powerful team programming environment that allows several programmers to share the same repository.

The Workspace

The workspace is stored as a single file (`IDE.icx`) and contains the program elements that you are currently working with as well as your preferences and working environment. You use the IDE to manage the workspace.

Whenever you close the last window of a VisualAge for Java session or select **File**→**Exit** from the Workbench menu bar, VisualAge for Java displays a dialog that prompts you to confirm the save of your workspace. If you click the **Cancel** button, Visual for Java does not exit and you stay in the Workbench.

The Repository

The repository is the database of your program elements. Each time you create or save a program element, it is stored in the repository as well as in the workspace. The repository contains all editions of all program elements, so it is usually quite a bit larger than your workspace. In addition, your workspace does not usually contain all the program elements that are available in the repository, because you do not necessarily need them for your current project. You can load program elements from the repository into your workspace, and you can delete program elements from your workspace. Figure 81 depicts the relationship between the repository and the workspace.

Information

Although VisualAge for Java manages all of your code, it does not manage resource files, such as images and sound clips. VisualAge for Java stores resource files in the file system, and you are responsible for managing them. You can find the resources for the MyProject project in the directory:

`\IBMVJava\ide\project_resources\MyProject.`

Version Control

VisualAge for Java supports a repository-based source control mechanism that allows you to keep track of all source code changes made over time. The VisualAge for Java source code repository is automatically updated each time you make a change to the source code. A history of all changes made to the source is kept within the repository, enabling you to back out any or all source code changes. The source control of your program elements is ruled by editions and versions.

Editions

A program element that you can modify is an *edition*. VisualAge for Java creates a new edition when you create or import a program element or when you modify a versioned program element.

Editions have a version number, which is a time stamp that indicates when the edition was created.

Versions

A *version* is a read-only edition of a program element, which is assigned a version number (such as 1.0). Whenever you version a program element, it is stored in the repository. You may want to version your program elements to:

- ☐ Take a snapshot of the state of your program elements to which you can revert if you want to abandon subsequent modifications.
- ☐ Have a baseline for code that you import from the file system and plan to modify.
- ☐ Freeze a finished program that you are ready to release.

Because you can have one edition and several versions of a program element, the repository also contains one edition and several versions, one of which can be loaded into the workspace at any time.

In Figure 81 the current workspace is loaded with the 1.5.1997 edition of a program element. This edition is also stored in the repository along with two different versions of the same element.

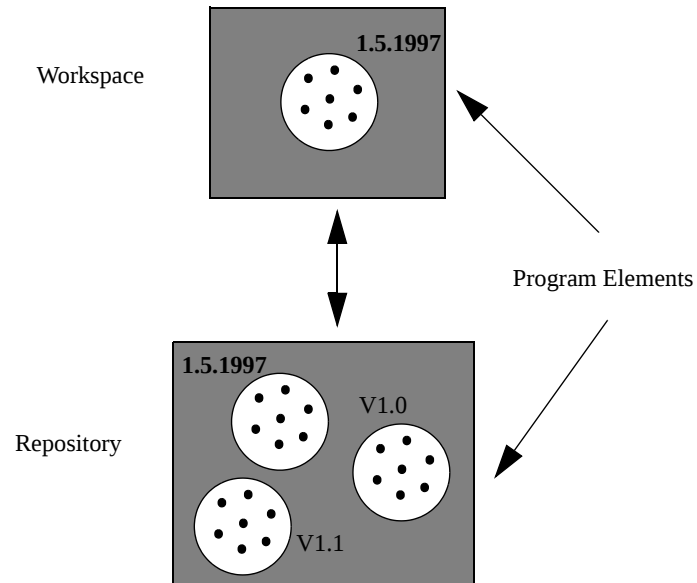


Figure 81. Program Elements in the Workspace and Repository

Read This



You should back up the repository and workspace files frequently. Also, you may want to save the original versions of these files when you install VisualAge for Java, so that you can return to the original state of the repository if required.

The workspace file (`IDE.icx`) is in the program directory with the VisualAge for Java programs. The repository (`ivj.dat`) is in the repository directory under the main VisualAge for Java IDE directory.

Versioning Program Elements

When you version a program element, you create a read-only edition of the element. Versions are also created of each nonversion program element contained within the element you are versioning. Already versioned elements contained within the element being versioned are not changed. For example, assume you have the following package and classes:

```
ch09 (7/24/97 1:57:43 PM)
  BankAccount 1.0
  CheckingAccount (7/24/97 1:57:43 PM)
```

The edition of ch09 and CheckingAccount is indicated in parentheses. The version of BankAccount is 1.0. When you version ch09, the CheckingAccount class will also be versioned, but BankAccount will not be versioned because it already is at Version 1.0.

To version an edition of a program element, select it and then click **Selected**→**Version** in the menu bar. The SmartGuide (Figure 82) appears.

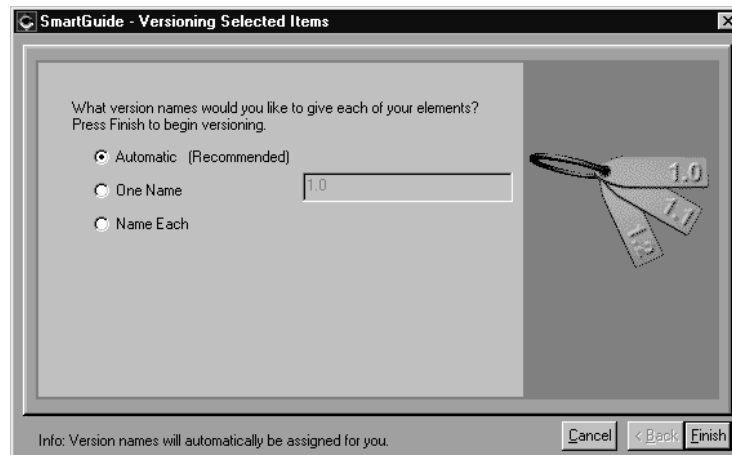


Figure 82. Versioning Program Elements

You have three options for naming your versioned elements:

- ☐ **Automatic naming:** Each program element to be versioned (that is, each non-versioned program element) is given a version name, which is an increment of its current version name.
- ☐ **One name:** All program elements to be versioned are given the same name.
- ☐ **Name each:** You are prompted to name each program element to be versioned.

VisualAge for Java versions the selected program element and its nonversioned elements and replaces the time stamps with the version names.

You do not explicitly version editions of methods. Every time you modify and save a method, or create a new one, within a versioned program element, VisualAge for Java creates a new edition of the program element with a time stamp that indicates the time that you saved it.

Managing Editions

Now that you are more familiar with editions and versions, let's see how you manage editions of your program elements.

Creating New Editions

When you add new program elements to the workspace or import Java code from the file system, VisualAge creates an edition of each program element. The edition is identified by a time stamp that indicates when VisualAge created the edition. You can click the **Hide Edition Names** and **Show Edition Names** Workbench toolbar buttons (the last two on the right) to hide or show the edition names with which you are currently working.

When you subsequently modify a program element that you have versioned, VisualAge for Java automatically creates a new edition of the element (marked with a time stamp). VisualAge for Java then writes an entry to the Log, stores the new edition in the repository, and replaces the versioned workspace edition with the new edition. The previous edition is still available from the repository.

Any versioned program elements that contain the element that was modified also become new editions. For example, if you modify a `TestClass` class that is at version 1.0, in a `testpkg` package at version 1.1:

- ☐ A new edition of `TestClass` is created with a time stamp.
- ☐ A new edition of `testpkg` is created with a time stamp.

Deleting a Program Element from the Workspace

When you delete a program element from the workspace, the edition is still in the repository, so you can reload it later. To delete a program element, select the element and choose **Selected→Delete** from the Workbench menu bar.

Replacing One Edition with Another

The workspace contains, at any one time, only one edition of a given program element, whereas the repository holds all editions of all program elements. You may want to replace the edition of a program element in the workspace with another edition so that you can abandon the modifications that you made and revert to another edition that resides in the repository.

To replace an edition of a program element with another edition, select the edition in the Workbench, select **Selected→Replace With→Another Edition** in the menu bar, and choose from the list that is displayed (Figure 83). The edition that is currently in the workspace is marked, by default, with an asterisk (*).

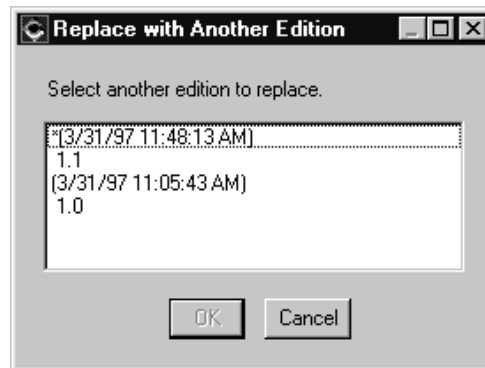


Figure 83. Loading Editions

To replace an edition with the edition from which it is derived, select the program element, and from the menu bar select **Selected→Replace With→Previous Edition**. When you replace an edition of a program element in the workspace, VisualAge also replaces the editions of the program elements that it contains, as required.

Loading Available Editions

To load a version of a program element that is not currently loaded in the workspace, use the **Selected** menu in the menu bar of the Workbench. The Selected menu contains some of the following items, depending on the selected item:

- ☐ **Add Project** (always available)
- ☐ **Add Package** (if a project is selected)
- ☐ **Add Class/Interface from Repository** (if a package is selected)
- ☐ **Add method from Repository** (if a class is selected)

For example, if you want to add a package from the repository to your project, select the project in which you want to add the package. Then, select **Selected→Add Package** in the menu bar to open the SmartGuide (Figure 84).

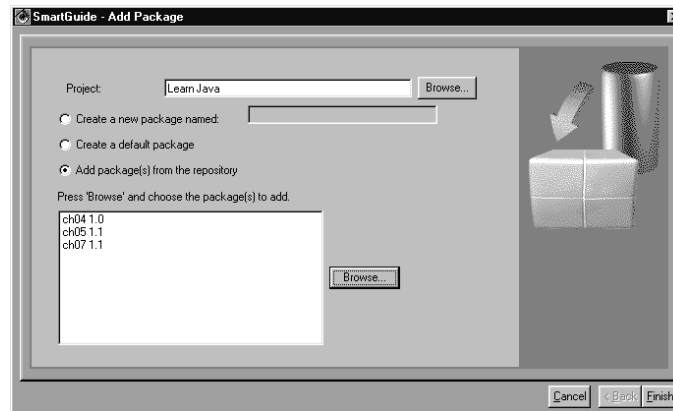


Figure 84. Import SmartGuide

Select **Add package(s) from the repository** and click the **Browse** button. VisualAge for Java shows you the available packages in the repository. Only the packages not currently in the workspace are displayed.

Using the Repository Explorer

The Repository Explorer is used to browse the contents of the repository. You open the Repository Explorer (Figure 85) by selecting **Window→Repository Explorer** in the menu bar.

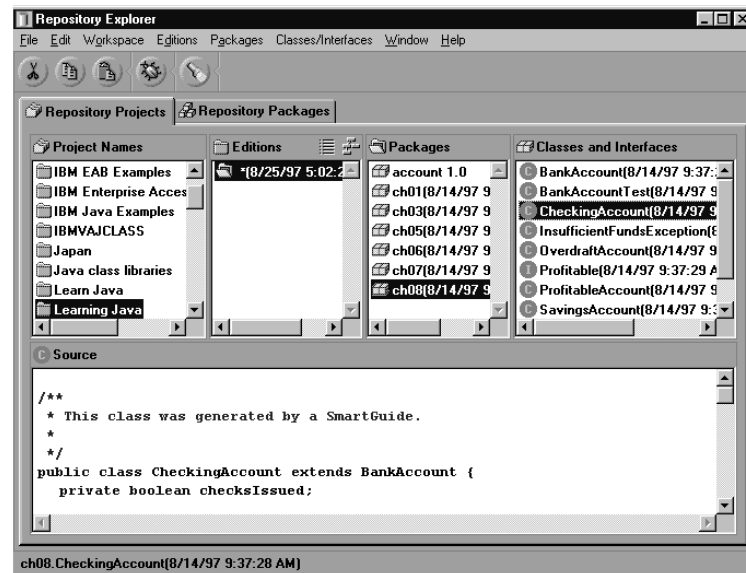


Figure 85. The Repository Explorer

To switch between the **Repository Projects** and **Repository Packages** views, click the tabs below the toolbar. The Repository Projects view shows all projects (Figure 85). To show the editions in the repository, just click the project, and the editions are listed in the list to the right of the project. Selecting an edition shows the packages in the selected edition, and selecting a package lists the classes in that package.

The Repository Packages view shows you all packages in the repository. Selecting a package lists all editions of the selected package, and selecting an edition shows you the classes and interfaces in that edition.

To see all the editions of a class, select the class, then select **Open** from the **Classes/Interfaces** menu bar and click on the **Editions in Repository** tab.

To compare two editions in either view, select two editions (if there is more than one), and select **Editions**→**Compare** in the menu bar. A window (Figure 86) appears where you can analyze the differences. You can also compare editions in the Workbench, using the **Selected**→**Compare With** menu item.

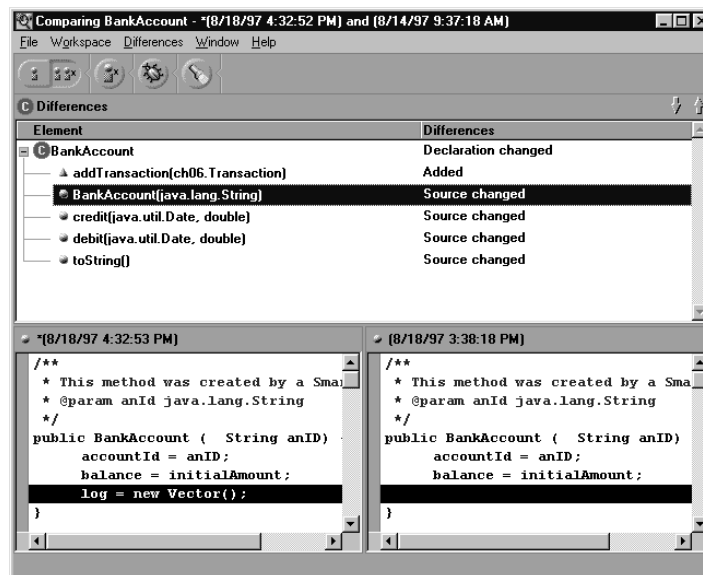


Figure 86. Comparing Two Editions

Searching for Program Elements

Another aspect of code management is finding all places where a particular class, field, or method is used or declared. VisualAge for Java search capabilities makes this task easy.

The Search Dialog

The Search Dialog allows you to search for references to, or declarations of, classes, methods, and fields. You access the Search Dialog (Figure 87) by selecting **Workspace**→**Search**.

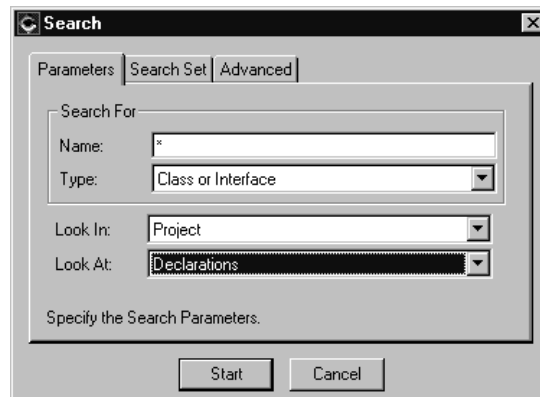


Figure 87. The Search Dialog

Customizing the Search

You can customize the search to retrieve the references and/or declarations of program elements (class, interface, method, constructor, field, or plain text) in various data sets such as:

- ☐ Workspace
- ☐ Project
- ☐ Package
- ☐ Hierarchy
- ☐ Search Set

The Search Set defines the list of projects from which you want to search your program elements (Figure 88).

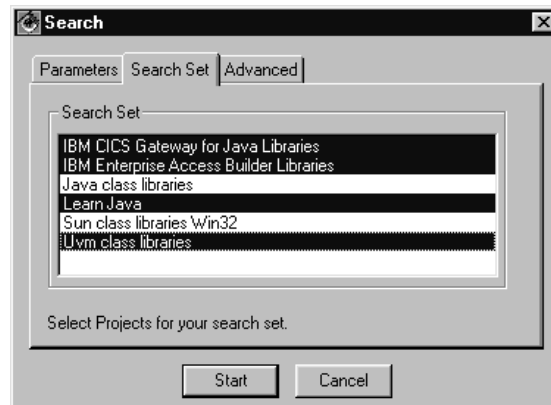


Figure 88. Search Set Customization

If you specify a plain text as your program element, the search text is used in combination with the results of the parameters to find the program elements containing the specific text.

Versioning the BankAccount

In this section you copy the package from Chapter 8 one last time. Now that you know how to use the VisualAge for Java versioning capabilities, you will use them to manage your code.

This time copy the `ch08` package to a new package named **account** in the Learn Java project. This package will contain the final version of the various classes you have written.

Now version the new package to create a baseline for the account code. In the Workbench, select the **account** package, and select **Selected→Version** in the menu bar. The SmartGuide prompts you for the names for editions. Select the **One Name** radio button, and type in *1.0* in the text field, and click the **Finish** push button. The Workbench window now displays your versioned package (Figure 89). If you do not see the edition names, click the **Show Edition Names** button in the toolbar.

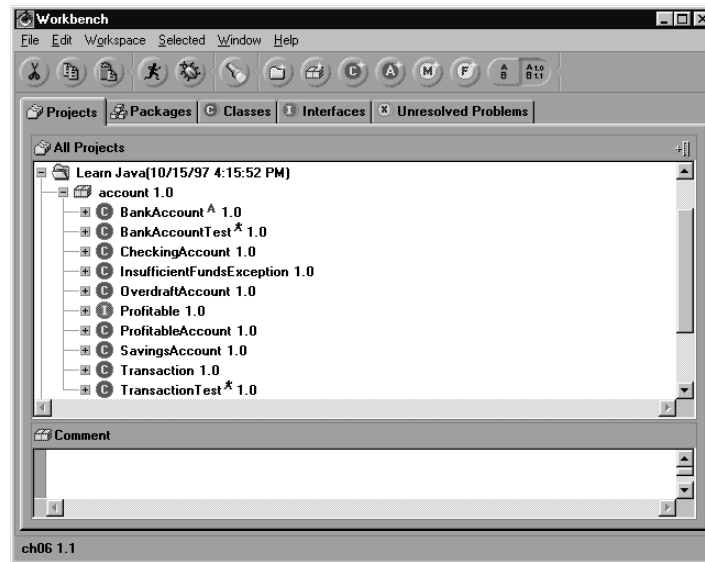


Figure 89. The Versioned Account Package

Now you have a version of your package to which you can always revert. To illustrate this, modify the `BankAccount` class by simply adding a comment in the class declaration, and save the method. VisualAge for Java displays a message similar to this in the Log window:

```
Made account.BankAccount(7/21/97 11:30:40 AM)
from account.BankAccount 1.0 in account(7/21/97 11:30:40 AM).
```

You now have a new edition of the `BankAccount` class. The new edition name is displayed in the Workbench (Figure 90). Click the **Show Edition Names** toolbar button if you do not see the name.

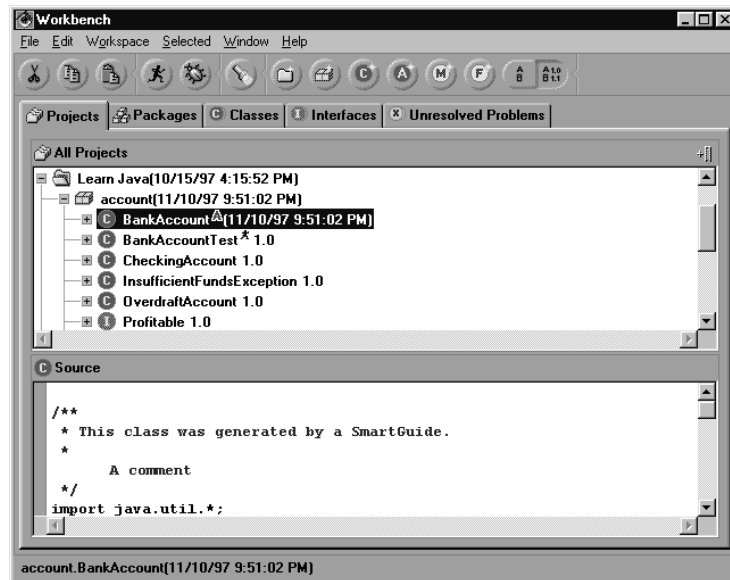


Figure 90. The New Edition of the BankAccount Class

You can load the previously versioned editions of the project, package, and class back into your workspace. Let's replace the BankAccount class. In the Workbench, select the BankAccount class, and select **Selected**→**Replace With**→**Another Edition** in the menu bar. VisualAge for Java displays a list of available editions in the repository. Select the **1.0** version you created previously and click the **OK** button. The selected edition is loaded, and the Workbench shows your package restored to the version shown in Figure 89.

Summary



VisualAge for Java stores your Java projects in the repository. The workspace is where you work on your current projects.

VisualAge for Java contains a powerful set of tools to control and manage versions of your packages, classes, interfaces, and methods. The tools are an integral part of the VisualAge for Java environment, so they are easy to use. The tools allow you to experiment with new versions of classes and easily revert to a version that has been tested and is known to work. You can also use the powerful search capabilities of VisualAge for Java to find classes, methods, and fields.

Part 2



Visual Building with JavaBeans

What makes people so excited about Java is the possibility of creating neat applets that can be downloaded from the Web and run inside a Web browser. These applets must be provided with a graphical user interface (GUI), however, so that the users can interact with them.

Up to now, you have developed Java applications without a GUI. These applications produced their output on standard output (std-out) and you could check it out by using the Console window. In this part we show you how to provide your Java applications with a GUI and explain how to write applets with the Abstract Window Toolkit of Java. We also describe the new Java component model, JavaBeans, and show you how you can take advantage of it through the Visual Composition Editor.

10

Building User Interfaces with Java

Java 1.1 contains a set of classes, called the *Abstract Windowing Toolkit (AWT)*, to build graphical user interfaces. In VisualAge for Java these classes are represented as components, and you will find out in Chapter 11, "Visual Programming and JavaBeans", just what that means.

In this chapter we introduce you to the use of AWT to hand code the GUI of your applets and applications. You can consider this chapter as your "purgatory" before you ascend to "heaven" in the next chapter. Indeed, in Chapter 11, you will learn how to visually compose your GUI and your whole application or applet and let the Visual Composition Editor of VisualAge for Java generate the code automatically for you. But for now, we want you to learn how to build your application GUI the hard way. Then it will be easier to understand what the Visual Composition Editor can do for you and the structure of the code it generates.

The examples presented in this chapter are not directly related to the ATM sample application that we have described so far. Not to worry, though; you will meet your ATM sample again in Chapter 11.

The Abstract Windowing Toolkit

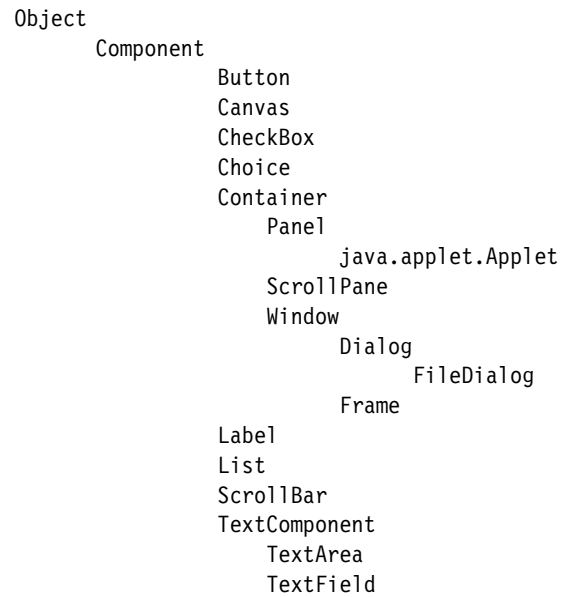
AWT is a set of classes that helps you develop the GUI of Java applets and applications. The original design goal was to provide programmers with a way of building a GUI that looks good on all platforms. A first version of AWT has been made available with JDK 1.0. This first attempt had many limitations and a programming model that was awkward to use and not very object-oriented. The new AWT provided with JDK 1.1 removes many of the limitations of the Java 1.0 AWT and introduces a new component programming model based on *JavaBeans*. The programming model eases the creation of GUI when you use visual programming tools such as the Composition Editor of VisualAge for Java. However, the new AWT still supports code that you developed using the original AWT.

In this chapter we show you how to program GUIs for your applications and how you can use the “old” AWT event model to make your applications react to events. Then, we show you how you can handle event events in your code by using the new event model of Java AWT 1.1. This section serves as a transition to introduce the JavaBeans model, which forms the foundation of the Visual Composition Editor that we describe in Chapter 11. We explain how you can use the layout managers to control the placement of your controls. We conclude by showing you how to provide menus to your applications and applets.

This chapter is not an exhaustive tutorial on how to use ATW. Rather, it will get you started and provide the key concepts to let you go further. For more information, make sure to use the JDK documentation.

The GUI Components

In Java, the term used to refer to various GUI building blocks is *components*. The classes for the components are divided into two hierarchies, one for the basic window components and one for menus. The classes are contained in the *java.awt* package. The window component hierarchy contains the following classes (Figure 91):

**Figure 91.** AWT Class Hierarchy

The **Component** class is the root of this hierarchy. It contains important methods to manipulate components. Table 6 lists some of the most important methods.

Table 6. Components Methods

Purpose	Method(s)
Show or hide a component	<code>setVisible(boolean)</code>
Paint the component	<code>paint(Graphics)</code> <code>paintAll(Graphics)</code> <code>repaint()</code>
Enable or disable a component	<code>setEnabled(boolean)</code>
Print the component	<code>print(Graphics)</code> <code>printAll(Graphics)</code>
Event processing	<code>processComponentEvent(ComponentEvent)</code> <code>processEvent(AWTEvent)</code>

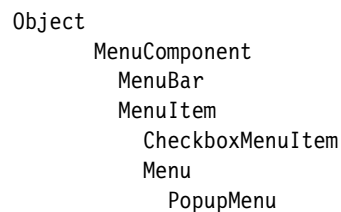
Table 6. Components Methods

Purpose	Method(s)
Set the bounding rectangle	setBounds(Rectangle) setBounds(x,y,width,height)
Set the size of a component	setSize(Dimension) setSize(int,int)
Set the layout manager	setLayout(LayoutManager)
Set the font	setFont(Font)
Update a component	update(Graphics)

In Java, components are placed inside *containers* such as Panel, Applet, Window or Frame. The Container class has a set of methods to add and remove components to or from containers. These methods include *add(Component)* and *remove(Component)*.

Every user interface implemented in Java contains at least one container, where you can place the other components. It is also possible to have containers within containers, so the user interface in fact forms a treelike hierarchy.

The MenuComponent and its subclasses handle the menus. Figure 92 shows the class hierarchy.

**Figure 92.** The MenuComponent and Subclasses

For additional information about the methods and responsibilities of each class, refer to the JDK 1.1 documentation. You can also use the Workbench to browse the classes and their source code.

Applets and Applications Revisited

By now you probably know that to create a stand-alone application that does not need a GUI you can use any class you like, but you must implement the main method:

```
public static void main(String argv[]){ // Some code here }
```

The main method is invoked by the Java interpreter first and then passed any command line arguments. Interactions with users are handled through the Console window, which provides a dialog window with stdin and stdout.

Something you may have not figured out yet is that when you create an applet you have no choice: you must provide it with a GUI! Indeed, because your applet is downloaded from the Internet to run inside your Java-enabled Web browser, you must create a GUI to let the user interact with the applet through the browser.

Applets

AWT provides applets and applications with a framework that produces the basic application or applet behavior needed in most cases. To customize this basic behavior, your application just has to override the methods of interest. When you build applets, your main class derives from `Applet`, which derives from `Panel`. Because `Applet` is a subclass of the `Panel` class, applets can contain components like text fields and buttons, just as any container can. Applets do not have to create a window in which to display themselves, because they usually just display themselves within the browser window.

Applets have a sequence of methods invoked by the browser or appletviewer when they are instantiated. The `Applet` class implements the following methods to facilitate communication between the browser and its containing application:

- ❑ `init()` is called by the browser or applet viewer to inform this applet that it has been loaded into the system.
- ❑ `start()` is called by the browser or applet viewer to inform this applet that it should start its execution. It is called after the `init` method and each time the applet is revisited in a Web page.
- ❑ `stop()` is called by the browser or applet viewer to inform this applet that it should stop its execution. It is called when the Web page that contains this applet has been replaced by another page and just before the applet is to be destroyed.

- ❑ `destroy()` is called by the browser or applet viewer to inform this applet that it is being reclaimed and should destroy any resources it has allocated.

Figure 93 illustrates the lifecycle of an applet. To customize the behavior of your applet, you customize the `init()`, `start()`, `stop()`, `paint()`, and `destroy()` methods that are called at the appropriate time by the application framework's control mechanism.

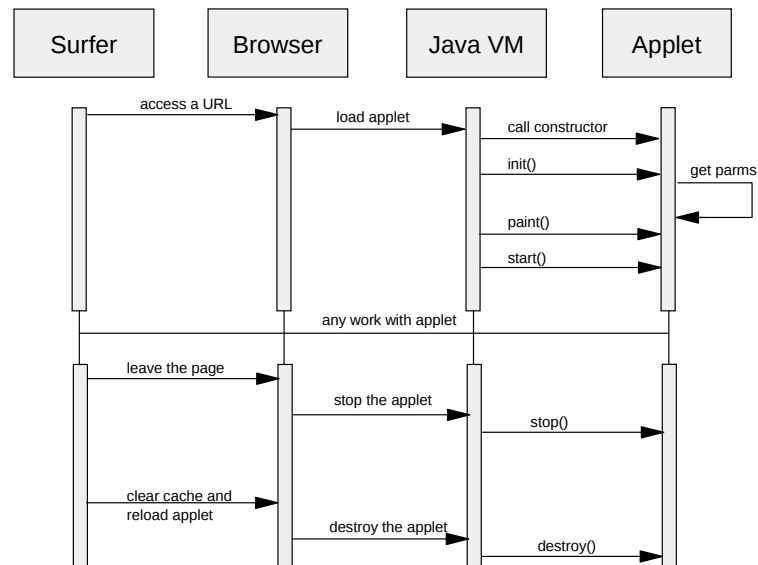


Figure 93. Lifecycle of an Applet

Applications

Applications that require a user interface, unlike applets, have to create their own windows because they are not displayed by a Web browser. This is one of the purposes of the `main()` method—to create the user interface and open the window.

Tip



It is possible to modify an applet so that it can also be run as an application. Simply provide a `main()` method to create a frame that can be used to display the applet.

Creating User Interfaces

When using VisualAge for Java, you have two options for creating the user interface of your applets or your applications: you can manually code the user interface by using the AWT library and the IDE tools, or you can use the Visual Composition Editor to automatically generate the code for you.

In this chapter we show you how to program an interface the “hard way,” by typing the code in the editor. In Chapter 11, we show how you can use the Visual Composition Editor to create most of the code for you.

Container Classes

To create an applet, you subclass the Applet class which inherits from Panel. To create an application with a user interface, you choose a subclass of Container that suits your need and create your own subclass of that class. Then you place other components within that container, using the add method in the main() method. Most of the time you end up subclassing the Dialog or the Frame class. Both classes inherit from the Window class, which enables you to create windows that sport titles, resize handles and menu bars. Dialog is a more limited windows for dialog boxes and can be made nonresizable and modal.

Coding Your First User Interface

To illustrate how to use the Frame, TextField, and Button classes, let's build an AWT application that displays an entry field and a push button in a window (Figure 94):

1. Create a new package, ch10, in the Learn Java project.
2. Select the newly created package and start the Create Class SmartGuide to create Sample1.
3. Choose java.awt.Frame for the superclass and make sure you select the **Write source code for the class** radio button.
4. Press the **Next>** push button to access the Attributes page. In the Attributes page, add java.awt.* to the list of packages to be imported and check the main(String[]) method to be generated.
5. Click the **Finish** push button to create your class.
6. Using the Create Field SmartGuide, add a field b1 of type java.awt.Button and a t1 field of type java.awt.TextField.

7. Modify the default constructor and the main method of your class:

```
import java.awt.*;

public class Sample1 extends java.awt.Frame {
    Button b1;
    TextField t1;
    public Sample1() {
        this.setLayout(null);
        t1 = new TextField("I am a text field");
        this.add(t1);
        t1.setBounds(50,50,150,30);
        b1 = new Button("Press to Exit");
        this.add(b1);
        b1.setBounds(50,100,150,50);
        this.setBounds(10,10,250,250);
        this.setVisible(true);
    }
    public static void main (String args[]) {
        new Sample1();
    }
}
```

The above example creates a window (Frame) with an entry field (TextField) and a pushbutton (Button). The Frame class enables you to create nonmodal windows for your application. Frames can also be created from an applet to create windows that are independent of the browser window containing the applet.

AWT provides other controls, some of which you will discover in this chapter. For a more thorough description of all the available controls, you can refer to the JDK documentation.

To control the position of components inside of a container, Java AWT provides layout manager classes that we describe in “Controlling Layout” on page 214. Because you want to control the positions of your components without the help of a layout manager, you call the `setLayout` method with a null parameter to disable the default layout manager. You then create an instance of a component, add it to the container, using the `add` method, and set the size and the position within the container, using the `setBounds` method. Finally, you call the `setVisible` method to show your window (Figure 94).

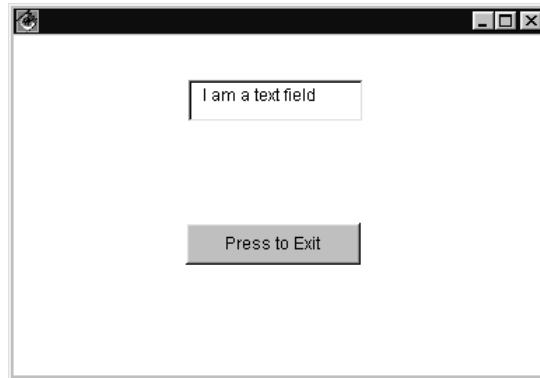


Figure 94. Placing Components in a Container

At this time, the window cannot close itself because you have not implemented any code to handle events that the window is supposed to handle. In the next section we show you how to handle user events, but for now you need to use the debugger to close your window:

1. Start the debugger by clicking the debugger icon in the Workbench toolbar.
2. Select the thread that runs your application.
3. Once the debugger has loaded the thread click the **Terminate** button on the debugger toolbar to terminate the thread.
4. Close the debugger.

Handling Events

To close the window of the previous example, you must capture the button pressed event and tie it to code that closes the window. Tying events to code is the basis of event-based programming and is the way the Java AWT 1.0 event model handles events. With AWT 1.1, a new event model, called a *delegation model*, overcomes some of the design limitations of AWT 1.0. Let's first use the Java AWT 1.0 event model to close the window.

Handling Events with Java AWT 1.0

You may wonder why you should use the “old” event model to program your applications or applets instead of the new one. Well, you should not! Sometimes however, you have to work with legacy Java code, which uses the old event model.

Because Java is an object-oriented language, you might expect that Java AWT 1.0 provides an object-oriented approach to handling events in your application. For example, you could build your own button that would inherit from the `Button` class and override a “button pressed” method to react accordingly. You could also have a generic event class that would contain a method for each event to which you want your application to respond.

Java AWT 1.0 does not use an object-oriented approach. Instead it uses a giant cascade of if statements to match the event with the object that was the event target and execute the corresponding code if a match exists. This cascade of if statements is coded inside of the `action()` method of your applet or application that you override to code the event handler.

The action() Method

The `action()` method takes two parameters: an event object, which you use to check its target by calling its `target` field, and an object, which constitutes extra information passed by the component that is generating the event. On completion, the `action()` method returns a boolean that indicates whether the event has been handled.

Let’s modify our first sample by adding an `action()` method to close the application when the button is pressed:

1. Add an `action()` method, using the SmartGuide. The method signature is: *public boolean action(Event evt, Object arg)*.
2. Modify the `action()` code:

```
public boolean action(Event evt, Object arg) {
    if (evt.target.equals(b1) {;
        dispose();
    }
    else {
        return super.action(evt, arg);
    }
    return true;
}
```

In the `action()` method, you test the `target` field of the event received against the `b1` button. If the button has been clicked, you close the window and release its resources by calling the `dispose()` method. A `true` value is returned at the end of the if statement. Otherwise, you return the boolean returned by calling the `action()` method of the superclass.

The `handleEvent()` Method

Some typical controls, such as buttons, checkboxes, and drop-down lists, have a “standard action” they can handle that causes the `action()` method to be called with the appropriate event object. For example, with a `List` object, the `action()` method is called when you double-click on one of its elements. If you want to handle a single-click event, you need more control and have to override the `handleEvent()` method that gets called first for every event. Anytime an event happens to a specific object, its `handleEvent()` method is called, and an event object is created and passed to the method. The default `handleEvent()` method, defined in the `Component` class, calls either the `action()` method or a similar method to indicate focus change, mouse, or keyboard activity. By testing the event object in the `handleEvent()` method, you gain more control over the `action()` method.

Deprecated Methods

The `action()` method is not the only method called by `handleEvent()`. Two other categories of methods are called to let you capture keyboard or mouse and focus events. To handle these events you simply override the specific method by providing your own code. These methods are defined in the base class `Component` and thus are available in all controls you might place in a container (see Table 7). These methods, called “deprecated methods,” are supported in Java AWT 1.1, but they might not be supported in the next version of the JDK. You may see legacy code using these methods, but you should use the new Java event model that we describe in the next section as much as possible.

Table 7. Deprecated Methods

Component Method	When It Is Called
<code>action(Event evt, Object what)</code>	A “typical” event occurs for this component.
<code>keyDown(Event evt, int key)</code>	A key is pressed while the component has the focus.
<code>keyUp(Event evt, int key)</code>	A key is released while the component has the focus.
<code>lostFocus(Event evt, int key)</code>	The focus has moved away from the target.
<code>gotFocus(Event evt, int key)</code>	The focus has moved into the target.

Table 7. Deprecated Methods

Component Method	When It Is Called
<code>mouseDown(Event evt, int x, int y)</code>	A mouse down has occurred over the component at coordinates x, y.
<code>mouseUp(Event evt, int x, int y)</code>	A mouse up has occurred over the component at coordinates x, y.
<code>mouseMove(Event evt, int x, int y)</code>	The mouse has moved while it is over the component.
<code>mouseDrag(Event evt, int x, int y)</code>	The mouse is being dragged while it is over the component.
<code>mouseEnter(Event evt, int x, int y)</code>	The mouse has just gotten inside the component.
<code>mouseExit(Event evt, int x, int y)</code>	The mouse has just moved away from the component.

Now that you know how to handle events with the Java AWT 1.0 event model, let's see how you should code your example using the new event model provided with AWT 1.1.

Handling Events with Java AWT 1.1

AWT 1.1 contains many changes. Among them is the new event model, which helps you code your applications by conforming to the well-known model/view/controller (MVC) design pattern.

The MVC Design Pattern

MVC was first introduced in Smalltalk-80 for building user interfaces. The MVC triad consists of three interrelated components: the *model* consists of the application objects, the *view* is the representation of the application objects to the user through the user interface, and the *controller* defines the way the user interface reacts to user input.

MVC separates model objects, which closely relate to the application business domain and should be unique for the business domain, from the view objects, which provide information representation to the user and can be multiple for the same model. By having your code conform to this model-view separation, you facilitate its maintenance because the stable code related to the application business does not get mixed up with less stable code related to the user interface.

Systems that supports the MVC application design must provide some sort of notification framework that plays the controller role to facilitate communication between views and models. The name of this notification framework and its model may differ in each system but its general principles remain the same.

Java AWT 1.1 Notification Framework

Instead of the non-object-oriented cascaded if statements of the old AWT event model, the new event model approach considers objects as *sources* and *listeners* of events. A source object can *fire* an event. Each type of event is an instance of a specific class. When an event is fired, it is received by one or more listeners, which act on that event. These listeners register with the source to receive the event. With this approach the source of the event and the place where the event is handled may be separate and more suitable to a model-view design of the application.

Event Sources and Listeners

Each event listener is an instance of a class that implements a specific type of listener interface. The code implemented by the listener for this interface is called when the event is fired. A listener is registered with a source object by calling an `addXXXListener()` method in the source object, where XXX represents the type of event listened for. For example, a listener registers with a button for the Action event by calling the `addActionListener()` method of the button.

Inner classes (which are classes defined inside other classes (see “Inner Classes” on page 118) are a useful feature of JDK 1.1 and later for logically grouping the listener classes inside the GUI or business logic classes. In addition, an inner class object keeps a handle on its parent object, which is very useful for calling across classes and system boundaries. However, inner classes and anonymous classes, which can be seen as an extension of inner classes, are not supported in IBM VisualAge for Java 1.0. Importing Java source that uses nested classes results in a parse error. Importing .class files that use nested classes works correctly. In this latter case, the classes are not easily accessible from the Java source because the only way to refer to the nested class is by its mangled name. Anonymous classes or inner classes are not distinguishable from regularly defined classes; they are treated as top-level classes with exotic names. Until support is provided, the best way to work with Java 1.1 nested classes is to compile them with JDK and import them as .class files.

Nevertheless, you can create separate listener classes without any problem and this is the option we choose for the rest of this chapter.

Modifying the Sample Code

If you rewrite the first sample to handle the “button pressed” event of your button and close the applet’s frame, you have to define a B1 class that implements the ActionListener interface. This interface must be implemented in order to respond to an ActionEvent, which is the clicked event of the push button. The closing of the window is delegated by the button to a B1 object. The interface contains only one method, actionPerformed(), which must be overridden to close the window:

```
import java.awt.*;

public class Sample2 extends java.awt.Frame {
    Button b1;
    TextField t1;

    public Sample2() {
        this.setLayout(null);
        t1 = new TextField("I am a text field");
        this.add(t1);
        t1.setBounds(50,50,150,30);
        b1 = new Button("Press to Exit");
        this.add(b1);
        b1.setBounds(50,100,150,50);
        b1.addActionListener(new B1()); // events sent
            // to a B1 object
        this.setBounds(10,10,250,250);
        this.setVisible(true);
    }

    public static void main (String args[]) {
        new Sample2();
    }
}

import java.awt.*;
import java.awt.event.*;
public class B1 implements ActionListener {
    public void actionPerformed (ActionEvent anEvent) {
        ((Frame)((Button)anEvent.getSource()).getParent()).
            dispose();
        return;
    }
}
```


Implementing Listeners in the Main Class

Another approach to the previous code, where the listener is coded in a separate class, consists of implementing the various listeners in the main class, typically an Applet or a Frame class:

```
import java.awt.*;
import java.awt.event.*;

public class Sample3 extends java.awt.Frame
    implements ActionListener{
    Button b1;
    TextField t1;

    public Sample3() {
        this.setLayout(null);
        t1 = new TextField("I am a text field");
        this.add(t1);
        t1.setBounds(50,50,150,30);
        b1 = new Button("Press to Exit");
        this.add(b1);
        b1.setBounds(50,100,150,50);
        b1.addActionListener(this); // events sent
            // to this (frame)
        this.setBounds(10,10,250,250);
        this.setVisible(true);
    }

    public void actionPerformed (ActionEvent anEvent) {
        dispose();
        return;
    }

    public static void main (String args[]) {
        new Sample3();
    }
}
```

This approach creates fewer classes and allows simpler code for the `actionPerformed()` action because the main class is the listener itself. However, the code is much less modular, and you must be able to detect which source creates the event with a cascading if statement in the `actionPerformed()` action in order to react accordingly. Notice that in your code, because the listener registered with only one source of event, the `b1` button, there is no need to check the event source in the `actionPerformed()` action.

As you will see, the Visual Composition Editor uses this approach to generate your application's source code.

Listener Types

When the listener interface contains only one method, you only have to implement that method. When the listener interface contains several methods, you have to implement each of them even if you decide that some method should not do anything. For example, when creating an application, you should always provide a `WindowListener` to the `Frame` in order to call the `System.exit(0)` to exit the application when you get the `WindowClosing()` event. In case of `WindowListener`, you have to implement all the abstract methods:

- ❑ `public void windowClosed(WindowEvent anEvent)`
- ❑ `public void windowClosing(WindowEvent anEvent)`
- ❑ `public void windowIconified(WindowEvent anEvent)`
- ❑ `public void windowDeiconified(WindowEvent anEvent)`
- ❑ `public void windowOpened(WindowEvent anEvent)`
- ❑ `public void windowActivated(WindowEvent anEvent)`
- ❑ `public void windowDeactivated(WindowEvent anEvent)`

Let's modify `Sample3` to write a message in the `t1` text field when the `b1` button is clicked and to handle the closing of the window. This time, the main class implements two interfaces: `ActionListener` and `WindowListener`:

```
import java.awt.*;
import java.awt.event.*;
public class Sample4 extends java.awt.Frame
    implements ActionListener, WindowListener {
    Button b1;
    TextField t1;

    public Sample4() {
        this.setLayout(null);
        t1 = new TextField("I am a text field");
        this.add(t1);
        t1.setBounds(50,50,150,30);
        b1 = new Button("Press to Exit");
        this.add(b1);
        b1.setBounds(50,100,150,50);
        b1.addActionListener(this);
        this.setBounds(10,10,250,250);
        this.addWindowListener(this);
        this.setVisible(true);
    }

    public void actionPerformed (ActionEvent anEvent) {
        if (anEvent.getSource() == b1)
            t1.setText(anEvent.getSource().toString());
        return;
    }
}
```

```
public static void main (String args[]) {
    new Sample4();
}
}
public void windowClosing (WindowEvent anEvent) {
    System.exit(0);
    return;
}
public void windowClosed(WindowEvent anEvent) {
    return;
}
public void windowDeiconified(WindowEvent anEvent) {
    return;
}
public void windowIconified(WindowEvent anEvent) {
    return;
}
public void windowOpened(WindowEvent anEvent) {
    return;
}
public void windowActivated(WindowEvent anEvent) {
    return;
}
public void windowDeactivated(WindowEvent anEvent) {
    return;
}
}
```

Different listeners are provided to let you handle major events such as resize, focus, selection, mouse, or keyboard events. Refer to the JDK1.1 reference for more information about those listeners.

Listener Adapters

When you create a listener class that implements a listener interface with several methods, it is always painful to implement methods you do not intend to use in your code. To prevent you from writing the implementation of these methods, each of the listener interfaces that have more than one method are provided with adapter classes. Each adapter provides a default behavior for each abstract method. Then, all you need to do is make your listener class inherit from the adapter and override only those methods you need to change.

Controlling Layout

Now that our example can handle both window events and action events, let's add some more components to the window.

You could of course use absolute positioning as you used in the first version of the example (Figure 94), but usually this kind of positioning works well only if every user of the class has a display with the same resolution and same fonts as the developer. In fact, what you need is a way of handling different display properties, and that is where layout managers come to the rescue.

Java uses layout managers to decide how the components are positioned on the basis of the order in which you add them in their container. The choice of a layout manager for your container directly impacts the size, shape, and placement of your components within that container. In addition, layout managers adapt dynamically to the dimensions of the container, solving many problems related to display resolution or font changes.

The classes implementing the layout managers are:

- ☐ `FlowLayout`
- ☐ `BorderLayout`
- ☐ `CardLayout`
- ☐ `GridLayout`
- ☐ `GridBagLayout`

Both applications and applets are derived from the `Container` class, which contains and displays objects, which are instances of the `Component` class. A `Container` is itself derived from `Component` to let you embed `Container` objects inside `Container` objects. You associate a layout manager with a container by sending the `setLayout(LayoutManager)` message to your container.

When you add components to a container that uses a layout manager, you can choose among the following methods:

- ☐ `add(Component)`
- ☐ `add(Component, int)`
- ☐ `add(Component, Object)`
- ☐ `add(Component, Object, int)`

The `Object` parameter is a constraint that further specifies how the component should be placed. The `int` parameter can specify the index of the component, so you do not need to have the `add` statements in the same order in which your components appear in your window.

In the sections that follow, we illustrate the use of each layout manager. You can use the SmartGuide and the integrated editor of VisualAge for Java to create each example and test it in the IDE.

FlowLayout

FlowLayout arranges the components in one or more rows. The components are laid out one after another. If a component does not fit in a row, a new row is started. FlowLayout is the default layout manager for containers other than Frame or Window that use BorderLayout. The following is an example of an applet using FlowLayout:

```
import java.applet.*;
import java.awt.*;

public class FlowApplet extends Applet {
    public void init() {
        String label;
        // *** set the layout manager
        this.setLayout(new FlowLayout());
        // *** the above line is not actually needed (default)
        for(int i=0; i<10;i++){
            label = "Button " + new Integer(i).toString();
            this.add(new Button(label));
        }
    }
}
```

This code creates an applet with 10 buttons with numbered labels. How the applet looks depends on its width. Figure 95 shows an example of an applet putting a maximum of three components in each row. You can resize the window to see how the buttons are displayed evenly in the client area.

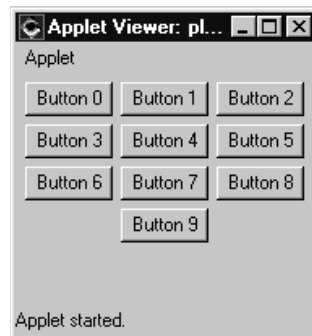


Figure 95. FlowLayout In Action

BorderLayout

`BorderLayout` organizes components, using constraints based on compass directions: *North*, *East*, *South* and *West*. The fifth “direction” is *Center*. You do not have to use all directions in your programs if you do not need them. Components are added to the container through the `add(Component aComponent, Object constraints)` method. Here is an example:

```
import java.applet.*;
import java.awt.*;

public class BorderApplet extends Applet {
    public void init() {
        this.setLayout(new BorderLayout());
        add(new Button("North"), "North");
        add(new Button("South"), "South");
        add(new Button("West"), "West");
        add(new Button("Center"), "Center");
        return;
    }
}
```

The resulting applet looks like that in Figure 96. You can resize the window and verify that the components are evenly displayed in the client area.

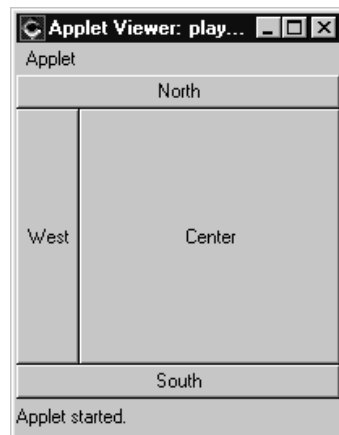


Figure 96. BorderLayout in Action

Notice that, because no component is put in *East*, the *Center* fills the remaining area.

CardLayout

CardLayout simulates a deck of cards, or a notebook, in the sense that you can put a number of pages on top of each other, and you can traverse the pages in several ways. Components are added to the container through the add method as with BorderLayout. Here is an example with two cards:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class CardApplet extends java.applet.Applet
    implements ActionListener {
    Button b1, b2;
    Panel p1, p2;
    TextField text;
    CardLayout layout;
    Image image;

    public void init() {
        p1 = new Panel();
        p2 = new Panel();
        p1.setLayout(new BorderLayout()); // set the layout manager
        b1 = new Button("To the second card");
        b2 = new Button("To the first card");
        b1.addActionListener(this);
        b2.addActionListener(this);
        layout = new CardLayout();
        this.setLayout(layout);
        this.add(p1,"First card"); // add two cards
        this.add(p2,"Second card");
        p1.add(b1,"North"); // first card includes button+image
        image = Toolkit.getDefaultToolkit().getImage("va.gif");
        p1.add(new ImageCanvas(image),"Center");
        p2.add(b2);
        p2.add(new Label("I guess I'm on the second card"));
        return;
    }
    // *** respond to button events ***
    public void actionPerformed (ActionEvent anEvent) {
        if (anEvent.getSource() == b1) // jump to the second card
            layout.show(this,"Second card");
        if (anEvent.getSource() == b2) // jump to the first card
            layout.show(this,"First card");
        return;
    }
}
```

The applet uses another class, `ImageCanvas`, to display an image on the first page. Here is the code for `ImageCanvas`:

```
import java.awt.*;
import java.awt.image.*;

class ImageCanvas extends Canvas {
    Image image;
    public ImageCanvas(Image anImage){// constructor
        image = anImage;           // takes Image as
    }                               // an argument
    public void paint(Graphics g) { // paint the image when needed
        Dimension d = size();      // center it inside parent
        int x = (d.width - image.getWidth(this)) / 2;
        int y = (d.height - image.getHeight(this)) / 2;
        g.drawImage(image, x, y, this);
    }
}
```

As you can see from the code, `CardApplet` actually uses two different layout managers internally: `BorderLayout` for the panel on the first card and `FlowLayout` for the panel on the second card. The above code results in an applet capable of switching between the two views as in Figure 97.

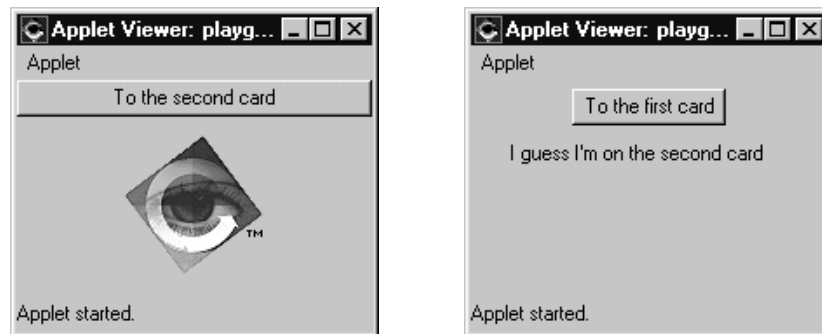


Figure 97. `CardLayout` in Action

GridLayout

`GridLayout` organizes components, using a predefined number of rows and columns. All cells in the resulting matrix are equal in size. It is possible to have some empty space, a gap, between cells. The components are added to cells starting from the top left corner, then going to the right. When a row is full, the next row is started from the left. Here is a listing of a simple integer calculator applet that uses both `GridLayout` and `BorderLayout`:

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class GridApplet extends java.applet.Applet
    implements ActionListener {
    TextField argument;
    TextField result;

    public void init() {
        Panel p1,p2;
        p1 = new Panel();

        result = new TextField("0");
        argument = new TextField("0");
        this.setLayout(new BorderLayout()); // set the layout manager
        this.add(argument,"North");        // argument field up
        this.add(result,"South");          // result field down
        this.add(p1,"Center");              // panel for keypad center
        this.setupKeypad(p1);
        return;
    }
    // *** build the keypad for the calculator ***
    private void setupKeypad(Panel aPanel) {
        char[] ops = {'+', '-', '*', '/', 'C', 'Q'};
        GridLayout layout = new GridLayout(4,4);
        Button b;
        aPanel.setLayout(layout);
        for (int i=0;i<10;i++){
            b = new Button(new Integer(i).toString());
            aPanel.add(b);
            b.addActionListener(this);
        };
        for (int j=0;j<6;j++) {
            b = new Button(new Character(ops[j]).toString());
            aPanel.add(b);
            b.addActionListener(this);
        };
        return;
    }
    // *** respond to button events ***
    public void actionPerformed (ActionEvent anEvent) {
        String label;
        // get the label of the button clicked
        label = ((Button)anEvent.getSource()).getLabel();
        if (label.compareTo("Q")== 0) // quit
            // System.exit(0); // if applet converted to an application
        if (label.compareTo("C")== 0){
            result.setText("0");
            argument.setText("");
        }
    }
}

```

```

        return;
    };
    // if a number 0..9, append to argument string
    if ((label.compareTo("0")>=0)&&(label.compareTo("9")<=0)){
        argument.setText(argument.getText() + label);
        return;
    };
    // convert the entry field values to int
    int target = new Integer(result.getText()).intValue();
    int source = new Integer(argument.getText()).intValue();
    // calculate
    if (label.compareTo("+") == 0)
        result.setText(new Integer(target + source).toString());
    if (label.compareTo("-") == 0)
        result.setText(new Integer(target - source).toString());
    if (label.compareTo("*") == 0)
        result.setText(new Integer(target * source).toString());
    if (label.compareTo("/") == 0)
        result.setText(new Integer(target / source).toString());
    argument.setText("");
    return;
}
}

```

When running the applet in the appletviewer, you should get the window displayed in Figure 98.

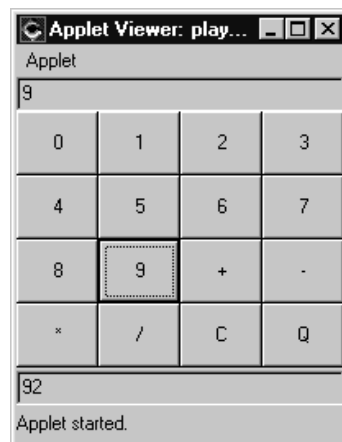


Figure 98. GridLayout in Action

The calculator has two entry fields and a panel arranged by using BorderLayout. The panel in the center uses GridLayout to organize the buttons. The calculator acts like an “HP”-style stack calculator. You enter the number first (displayed in the upper entry

field), then select the operator (+, -, *, /), and the operation is executed against the value in the lower entry field, which displays the result of the operation.

GridBagLayout

GridBagLayout is the most advanced and flexible, but also most complex to use, of the layout managers. The advantage of using GridBagLayout instead of GridLayout, which it resembles, is that the components do not have to be of the same size. You can use several types of constraints when attaching components to a container by using the GridBagConstraints class to define the attachments.

Let's first look at an example, a to-do list applet that you can use to maintain a list of to-do tasks.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class GridBagApplet extends java.applet.Applet
    implements ActionListener {
    Button add;
    Button remove;
    Button clear;
    TextField text;
    List list;

    public void init() {
        add = new Button(" Add ");
        remove = new Button("Remove");
        clear = new Button("Clear");
        text = new TextField("Things to do");
        list = new List();
        add.addActionListener(this);
        remove.addActionListener(this);
        clear.addActionListener(this);
        this.setLayout(new GridBagLayout());
        this.add(new Label("Add Item:"),
            this.constraints(0,0,1,1,0,0));
        this.add(text, this.constraints(1,0,1,1,1,0));
        this.add(new Label("To Do:"), this.constraints(0,1,1,1,0,0));
        this.add(list, this.constraints(0,2,2,2,1,1));
        this.add(add, this.constraints(3,0,1,1,0,0));
        this.add(remove, this.constraints(3,1,1,1,0,0));
        this.add(clear, this.constraints(3,2,1,2,0,1));
        return;
    }
}
```

```

private GridBagConstraints constraints
    (int x, int y, int w, int h, int wx, int wy) {
    GridBagConstraints cons = new GridBagConstraints();
    cons.gridx = x;
    cons.gridy = y;
    cons.gridwidth = w;
    cons.gridheight = h;
    cons.weightx = wx;
    cons.weighty = wy;
    cons.fill = GridBagConstraints.BOTH;
    cons.insets = new Insets(4,4,0,0);
    return cons;
}

public void actionPerformed (ActionEvent anEvent) {
    if(anEvent.getSource() == add) {    // add item
        list.addItem(text.getText());
    }
    if(anEvent.getSource() == remove) { // remove selected
        if (list.getSelectedItem() != null)
            list.remove(list.getSelectedItem());
    }
    if(anEvent.getSource() == clear)    // clear list
        list.removeAll();
    return;
}
}

```

This results in an applet like that in Figure 99.

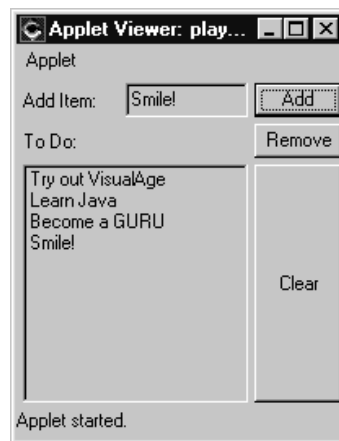


Figure 99. GridBagLayout in Action

In this example, you set the layout manager to `GridBagLayout` and then add the components, using the `add(Component, Constraints)` message. The constraints object is generated by a helper method `constraints(int, int, int, int, int, int)`. This method sets up and returns a `GridBagConstraints` object with appropriate values:

- ❑ **gridx**: starting column of the component (starts from 0)
- ❑ **gridy**: starting row of the component (starts from 0)
- ❑ **gridwidth**: the number of columns the component occupies
- ❑ **gridheight**: the number of rows the component occupies
- ❑ **weightx**: how the component should expand horizontally if stretched
- ❑ **weighty**: how the component should expand vertically if stretched
- ❑ **fill**: specifies how the component should occupy the cell
- ❑ **insets**: the space between components (bottom, left, top, right)

The example above uses a gap of four pixels between each component (`Insets(4,4,0,0)`), and each component fills the whole cell (`fill=GridBagConstraints.BOTH`). The listbox is expanded both vertically (`weighty=1.0`) and horizontally (`weightx=1.0`) in the same proportion if the window is stretched. The Clear button is expanded vertically, and the entry field, horizontally. All other components retain their size. Figure 100 explains the constraints of the example visually.

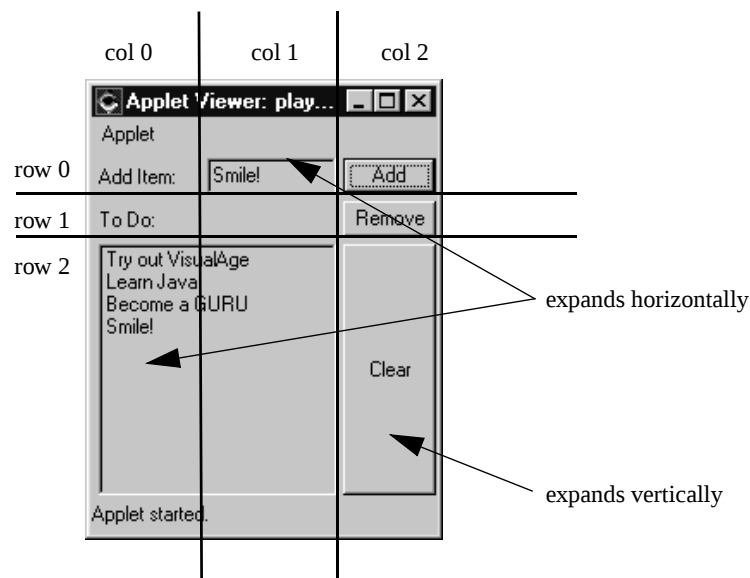


Figure 100. GridBagLayout Constraints

Combining Layout Managers

You have already seen examples, like the calculator, that use two different layout managers at the same time. You are free to use as many layout managers as you like with your classes and applets. However, if your windows become complex, the code for the classes becomes complex as well. This is when you should start thinking about splitting your windows into smaller parts, each in a separate class, and each possibly using a layout manager that best fits the need of the class.

Creating and Using Menus

A GUI application is seldom complete without a menu bar. The classes used to build menus are also included in the `java.awt` package. The menu-related classes include `MenuComponent`, `MenuItem`, `CheckboxMenuItem`, `PopupMenu`, `Menu`, and `MenuBar`. The last two are containers that organize noncontainer-type menus. Typically your menu contains a `MenuBar` object, which contains `Menu` objects. These, in turn, include `MenuItem` objects.

Using Menus with Applications

You attach a `MenuBar` object to a `Frame` object by sending it a `setMenuBar(MenuBar)` message. `Menu` objects and `MenuItem` objects are added to their containers through the `add()` method, as with any other containers.

To respond to menu events (when a menu item is selected), you must register an `ActionListener`. Here is a sample application with a menubar:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class MenuFrame extends Frame implements ActionListener{
    MenuBar bar;
    Menu fileMenu;
    Menu helpMenu;
    MenuItem mOpen;
    MenuItem mExit;
    MenuItem mContents;
    MenuItem mIndex;
    TextField text;
```

```

public MenuFrame() {
    this.setLayout(new BorderLayout());
    text = new TextField("I'll show what was selected");
    this.add(text,"South");
    this.buildMenu();
    this.setVisible(true);
    return;
}

public void buildMenu() {
    bar = new MenuBar(); // create a menubar
    this.setMenuBar(bar); // attach to this frame
    fileMenu = new Menu("File");
    helpMenu = new Menu("Help");
    bar.add(fileMenu); // add File menu
    bar.add(helpMenu); // add Help menu
    mOpen = new MenuItem("Open");
    mExit = new MenuItem("Exit");
    mContents = new MenuItem("Contents");
    mIndex = new MenuItem("Index");
    fileMenu.add(mOpen);
    fileMenu.add(mExit);
    helpMenu.add(mContents);
    helpMenu.add(mIndex);
    mOpen.addActionListener(this);
    mExit.addActionListener(this);
    mContents.addActionListener(this);
    mIndex.addActionListener(this);

    fileMenu.addActionListener(this);
    helpMenu.addActionListener(this);
    return;
}

public void actionPerformed(ActionEvent anEvent) {
    if (anEvent.getSource() == mExit)
        System.exit(0);
    text.setText(anEvent.getSource().toString());
    return;
}

public static void main (String args[]) {
    MenuFrame mf = new MenuFrame();
}
}

```

The above class implements the `ActionListener` interface. The menu actions are handled in the `actionPerformed` method, and the information about the menu selection is sent to the text field. The menu bar and its menu components are constructed in the `buildMenu` method. The result looks like the window in Figure 101.

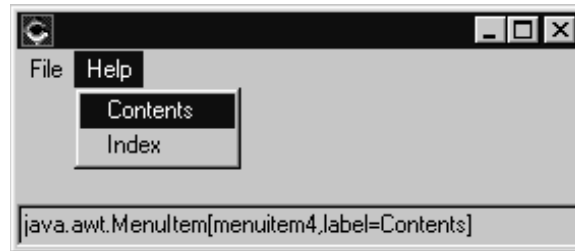


Figure 101. Frame with a Menubar

Using Menus with Applets

Because menu bars can only be attached to Frame objects, applets cannot have menu bars. Instead, you can use a pop-up menu, which you can attach to a component. Here is an applet that has a pop-up menu:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class PopupApplet extends Applet
    implements ActionListener, MouseListener {
    PopupMenu bar;
    TextField text;

    public void init() {
        super.init();
        text = new TextField("I'll show what was selected");
        this.setLayout(new BorderLayout());
        this.add(text,"South");
        Button b = new Button("Press");
        this.add(b,"North");
        b.addActionListener(this);
        this.addMouseListener(this);
        this.buildMenu();
        return;
    }

    public void buildMenu() {
        // build the popup menu
        bar = new PopupMenu();
        MenuItem oItem = new MenuItem("Open");
        MenuItem cItem = new MenuItem("Close");
        MenuItem eItem = new MenuItem("Exit");

        bar.add(oItem);
        bar.add(cItem);
        bar.add(eItem);
    }
}
```



```

        oItem.addActionListener(this); // send messages to this
        cItem.addActionListener(this);
        eItem.addActionListener(this);
        this.add(bar);
        return;
    }
    public void actionPerformed(ActionEvent e) {
        // show the action in the text field
        text.setText(e.getSource().toString());
    }
    public void mouseClicked(java.awt.event.MouseEvent e) {
        // show the popup where mouse was clicked
        // if (e.isPopupTrigger()) // should check the key pressed
        bar.show(this,e.getX(),e.getY());
    }
    public void mouseEntered(java.awt.event.MouseEvent e) {}
    public void mouseExited(java.awt.event.MouseEvent e) {}
    public void mousePressed(java.awt.event.MouseEvent e) {}
    public void mouseReleased(java.awt.event.MouseEvent e) {}
}

```

The resulting applet looks like that in Figure 102.

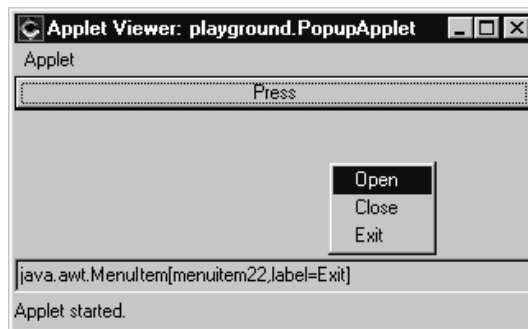


Figure 102. Applet with a Pop-up Menu

This applet listens to action events and mouse events. A pop-up menu is shown over the applet at the position of the mouse when the mouse is clicked. The `MouseEvent` class implements a method, *boolean isPopupTrigger()*, that can be used in a platform-independent way to check whether an event is indeed a pop-up menu request. All menu items send their action events to the applet, which shows the selected item in the text field.

Our example uses a pop-up menu only for the applet itself. It is also possible to create context-sensitive pop-up menus; you just have to create a pop-up menu for every component that wants one.

Pop-up menus are not restricted to applets. All other containers are free to use pop-up menus for stand-alone applications as well.

Now that you have spent some time in the AWT “purgatory,” you might want to rest a bit. This is exactly what we propose that you do in Chapter 11 where you will discover “programming heaven” using the JavaBeans component model and the Visual Composition Editor.

Summary



Java has a set of components for GUIs. These components are included in the `java.awt` package. The package includes user interaction components, such as buttons, text fields, lists, and menus. When you constructing user interfaces, you organize those components inside containers, such as windows, applets, frames, and panels. To make organizing and behavior control easier for programmers, Java introduces layout managers, which control, for example, how components behave when their parent window is resized.

11

Visual Programming and JavaBeans

In Chapter 10 you programmed your GUI the “hard way.” But there is light at the end of the tunnel! With the Java component model and the Visual Composition Editor, you can visually program not only the user interface of your applet or application, but the whole applet or application itself.

In this chapter we introduce you to visual programming and to the Java component model, JavaBeans. We show you how to work with JavaBeans in VisualAge for Java. You will also see more of the Java 1.1 event model as it relates to JavaBeans.

This chapter is not for bean developers who need to know everything possible about developing and deploying beans; it is more for developers who are using beans in the IBM VisualAge for Java environment.

Software Components

Despite the benefits of object-oriented development, implementing large systems can still be expensive. One way to reduce the cost is to reuse object implementations. Many companies would prefer to buy reliable reusable classes, creating classes only for functions specific to their business. This vision of constructing custom software using standard building blocks is called *construction from parts*. The building blocks themselves are popularly called parts or *components*. However, reuse is difficult to achieve when the class interfaces are too specific to the application for which they were originally developed and when no real specification is defined for the components you need to use.

Software components are to software what integrated circuits are to electronics. Basically, they encapsulate function and provide services based on a strict specification of their interface. Because of this specification, they can be used as “black boxes” and combined with other software components to build a full-blown application.

Software Components and Classes

Software components hide implementation, conform to interfaces and encapsulate data just like classes. In that sense components are also classes. But in addition to being classes, they also conform to a software component specification that makes them more reusable.

Benefits of Software Components

Creating and using software components can help a development organization build better applications faster. By having access to a reusable component base, programmers can quickly compose new applications from proven components. In addition, component architectures can help enforce a separation between the application model and its user interface, leading to a model-view design that is suitable for better maintenance.

Over time, class interface specifications called *component models* have been defined, such as ActiveX, OpenDoc, and JavaBeans to provide wider reuse. JavaBeans is the standard component model for the Java language and is the component model used by VisualAge for Java.

JavaBeans

JavaBeans is not a product, a language, or a development environment. It is a Java package (`java.beans`), and it is a class specification that describes how to use the classes and interfaces provided in the `java.bean` package to turn Java classes into software components called *beans*.

The JavaBeans specification includes the following definitions:

- ❑ **An event model:** Event models specify how a component sends messages to other objects without knowing the exact methods that the other object implements. Thus a component can be reused with a range of objects that have different interfaces.
- ❑ **Events, properties, and methods:** JavaBeans defines a component interface in terms of the events it can signal, its property values that can be read and set, and the methods it implements. This definition provides more structure to the interface of a component compared with a simple class interface, facilitating the use of tools such as the VisualAge Visual Composition Editor.
- ❑ **Introspection:** Introspection refers to the ability to discover programmatically the component interface for instances of a particular component class. Introspection enables the use of programs such as the Visual Composition Editor that can work with component instances at run time without having the details of the components programmed into them.

Beans can be visual components, such as a button or a list, that you use to build the user interface or *view* of your application. They can also be nonvisual components, such as bank account, that typically represent the business logic or *model* of your application.

To build an application or an applet using beans, you typically drag and drop the beans you want to use into a working area and wire them together. This visual programming approach does not require any real programming skills as long as you just reuse beans that have been previously created by someone else.

Visual Programming

The Visual Composition Editor enables you to create programs visually from existing beans.

VisualAge for Java provides user interface beans based on classes in the Java AWT package as well as nonvisual beans that you can reuse in your applications or applets.

The Visual Composition Editor is also extensible. It enables you to work with beans you create yourself and to include beans imported into the environment from other sources. In this chapter we show you how to use the Visual Composition Editor to create your own beans visually and then reuse them within another program you create with the Visual Composition Editor.

To build a program with the Visual Composition Editor, you draw a picture, using a canvas and a palette of icons representing reusable beans. The picture specifies the set of beans that implements the function of the larger program (or bean) you are creating. For visual beans such as user interface controls, the position of the controls relative to each other in the picture specifies how the controls will appear in the final program. For nonvisual beans such as database components, the position of the bean in the picture generally has no significance because the beans are not displayed at run time.

The Visual Composition Editor provides a sophisticated wiring capability to specify how components of the picture will interact to implement functions of the program. Using connections, you can graphically specify much of the behavior of an application and integrate custom code written in the Java language.

By now you must be eager to start programming visually, so let's start right away, using the Visual Composition Editor.

The Visual Composition Editor

The Visual Composition Editor helps you use simple beans to construct more complex beans. The new beans are assembled to create complex beans or applets and applications.

The Visual Composition Editor can manipulate two types of beans:

- ❑ **Visual beans**, which are subclasses of the `java.awt.Component` class. Examples are `List`, `Button` or `TextField`. They have a visual representation at run time.
- ❑ **Nonvisual beans** usually represent the business logic in your programs, for example, a `BankAccount` bean. Although they do not have a visual representation at run time, they are needed to implement the logic of your application.

Visual beans are represented in the tool as you would see them at runtime. Nonvisual beans appear as a puzzle icon. You manipulate visual and nonvisual beans with a set of tools that we describe in the following section.

First Contact

When opening a class browser on a visual bean, you have probably noticed the **Visual Composition** tab. If you select it, you switch to the Visual Composition Editor, which is your main tool for building your application visually (Figure 103).

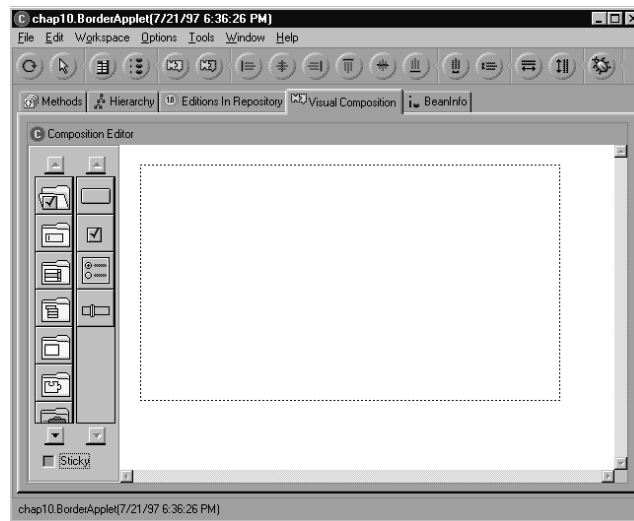


Figure 103. Visual Composition Editor

Let's have a look at the different components that comprise the Visual Composition Editor and see how you can use them to build a Java program.

The Class Browser Tabs

Although these are not strictly part of the Visual Composition Editor, the tabs shown in Figure 104 enable you to switch between the various pages of the class browser. They are:

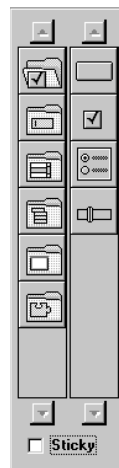
- ☐ **Methods:** displays the methods of the class
- ☐ **Hierarchy:** displays the class hierarchy of the class
- ☐ **Editions in Repository:** displays the editions of the class
- ☐ **Visual Composition:** the visual representation of the class

- ❑ **BeanInfo:** displays the bean information of the class. The BeanInfo page is discussed in “The BeanInfo Page” on page 252.



Figure 104. Class Browser Tabs

The Beans Palette



The Beans Palette is divided in two columns:

- ❑ The **categories** column on the left:

Beans are grouped together by function. The categories are: Buttons, Data Entry, Lists, Menus, Canvas, and Models.

- ❑ The **beans** column on the right:

When you select a category, the beans column is updated to show the beans of that category. Visual beans are mapped to their corresponding AWT controls and are used to build your application GUI. Nonvisual beans are used to code the application logic.

Figure 105. The Beans Palette

Notice the **Sticky** checkbox in Figure 105 . When you check it and you drop a part on the free-form surface, the pointer remains loaded with that bean. Thus, you can drop on the free-form surface several beans of the same type without reselecting the bean.

Tip



You can add to or modify the Beans Palette in several ways:

- ❑ Use the **Add To Palette...** menu item in the **File** menu to add new beans.
- ❑ Use the **Modify Palette** item in the **Options** menu. Using this item you can:
 - Add and remove categories
 - Add and remove beans

The Toolbar

The toolbar (Figure 106) provides you with a useful set of shortcuts to menu actions. You can test your code, edit the bean properties, list the beans located on the free-form surface, show or hide connections, show or hide the grid, snap to the grid, manipulate the beans (align, resize), and invoke the debugger.



Figure 106. The Toolbar

The Free-Form Surface

The free-form surface is the place where you assemble and wire the beans of your application (Figure 107).

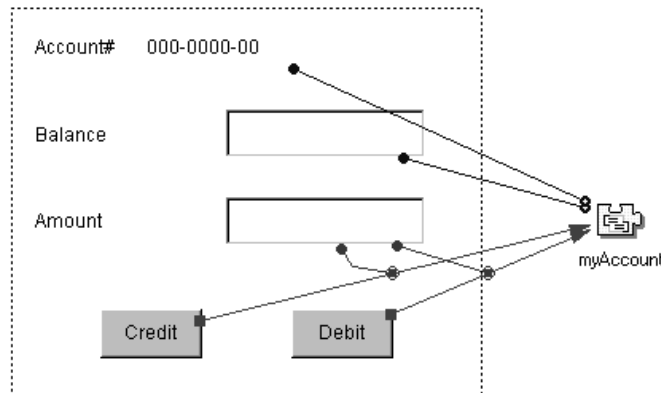


Figure 107. The Free-Form Surface

The free-form surface represents the *primary bean*, that is the bean you are editing in the class browser. Any beans you place on the free-form surface are private instance attributes of the class you are editing.

For example, suppose you visually create an applet, `MyApplet`, using the Visual Composition Editor as shown in Figure 107. By adding the `myAccount` nonvisual bean on the free-form surface, you literally tell the Visual Composition Editor to add a private instance attribute to the `MyApplet` bean. The type of the attribute is the type of `myAccount` and its name consists of the bean's name prefixed with `ivj:` *ivjmyAccount*.

If the primary bean has a visual representation, beans dropped onto the visual representation are shown visually at run-time. Beans added outside the visual representation are not added to the GUI of the primary bean.

The Visual Composition Editor, through visual feedbacks, prevents you from adding beans in the wrong place on the free-form surface.

Basic Bean Manipulations

There are two ways of adding a bean to the free-form surface: adding the bean from the palette or adding the bean by specifying its class name.

Adding a Bean from the Palette

To add a bean from the palette:

1. Select a category in the Palette. From the right column, select the part you want to add. Notice that the cursor turns to a cross, indicating that the selected bean is “loaded” onto the cursor and can be dropped onto the surface.
2. Move the cursor where you want to drop the bean. Click the left mouse button to drop the part on the free-form surface.

Adding a Bean by Specifying Its Class Name

To add a bean by specifying its class name:

1. From the **Options** menu, select **Add Bean...** (notice the key shortcut, CTRL-B).
2. The **Add Bean** dialog box appears (Figure 108).
3. Type the class name of your part in the first text field. Remember to use the fully qualified name (for example, `account.Vector`) or use the **Browse** button to find the class.
4. In the second text field, type the name of the bean (which should be a Java valid name, for example, `myAccount`). The name of the bean will be prefixed by *ivj* in the generated Java source code and shown on the free-form surface.
5. Select whether you are creating a class, a variable, or a serialized bean read from a serialization file.

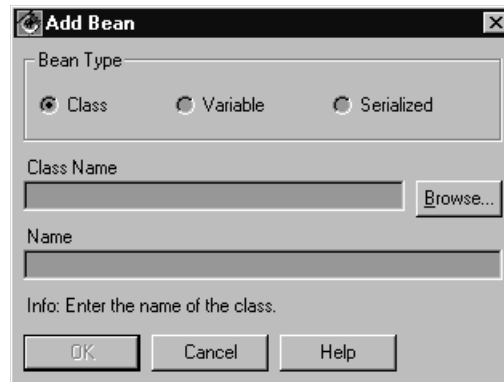


Figure 108. The Add Bean... Dialog

Information



In VisualAge for Java the beans that you drop on the free-form surface can be accessed from the primary bean from their accessor method. This accessor method constructs the bean the first time it is called. For example, if you have a bean named *myButton*, the first time *getMyButton()* is called, the button is constructed.

Customizing Beans

Once you have dropped a bean on the free-form surface, you can customize it by double-clicking the bean to open its property sheet. Using the property sheet, you can change properties exposed by the bean (see “Properties” on page 248).

Figure 109 shows the property sheet of a `TextField` bean. Notice the Show expert features checkbox, which lets you access internal properties of the bean.

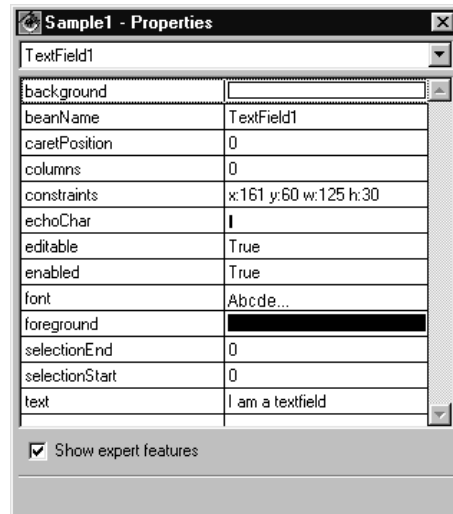


Figure 109. Property Sheet of a TextField Bean

Creating a Simple User Interface

To illustrate how you can use the Visual Composition Editor, let's create a simple user interface and then have a look at the code VisualAge for Java generates.

Visual Programming in Action

Here you rebuild the first example you wrote in Chapter 10:

1. Create a new package, *ch11*, in the Learn Java project.
2. Select the newly created package and start the Create Class SmartGuide to create *Sample1*.
3. Choose `java.awt.Frame` for the superclass and make sure this time you select the **Design the class visually** radio button.
4. Click the **Finish** push button to create your bean.

The Visual Composition Editor opens and displays the frame on the free-form surface (Figure 110).

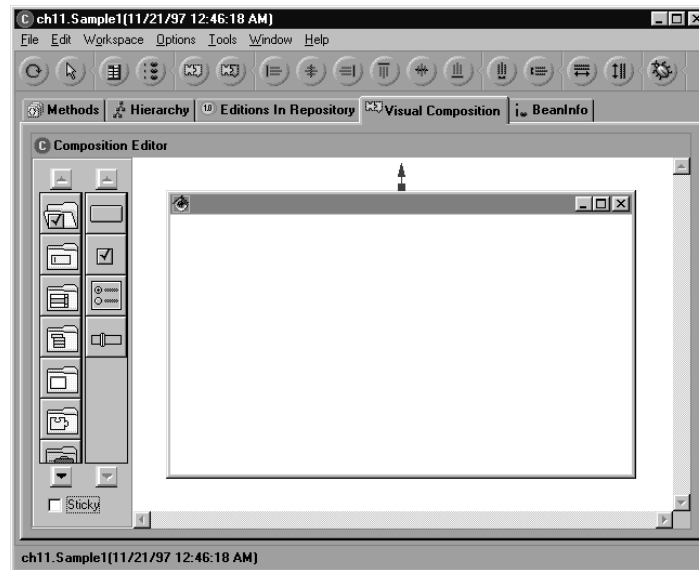


Figure 110. A Frame on the Free-Form Surface

Notice that a green connection is automatically added from the frame to the free-form surface. This connection is used to call the `dispose()` method of the `Frame` when the window is closed.

Now let's add a test field and a button to the frame.

1. Click on the **Data Entry** category and then click on the **TextField** bean on the palette.
2. Move the cursor over the visual representation of the frame on the free-form surface and click the left mouse button to drop the `TextField`.
3. Drop a **Button** from the **Buttons** category in the same way.

If you are unsure about the names of the categories or parts, click the left mouse button over the category or part and look at the description in the status line at the bottom of the page.

Let's add the text "I am a textfield" to the `TextField` and the text "Press to Exit" to the button.

To add the text to the `TextField`, select the `TextField` and then hold down the right mouse button and select **Open Settings** from the pop-up menu. Type "I am a textfield" to the right of the **text** property in the dialog and then click on **OK**.

To add the text to the button, simply hold down the ALT key and click with the left mouse button over the label on the button, then type in “I like to be pushed.” When you are finished, click anywhere else on the free-form surface. Your frame should look like that shown in Figure 111.

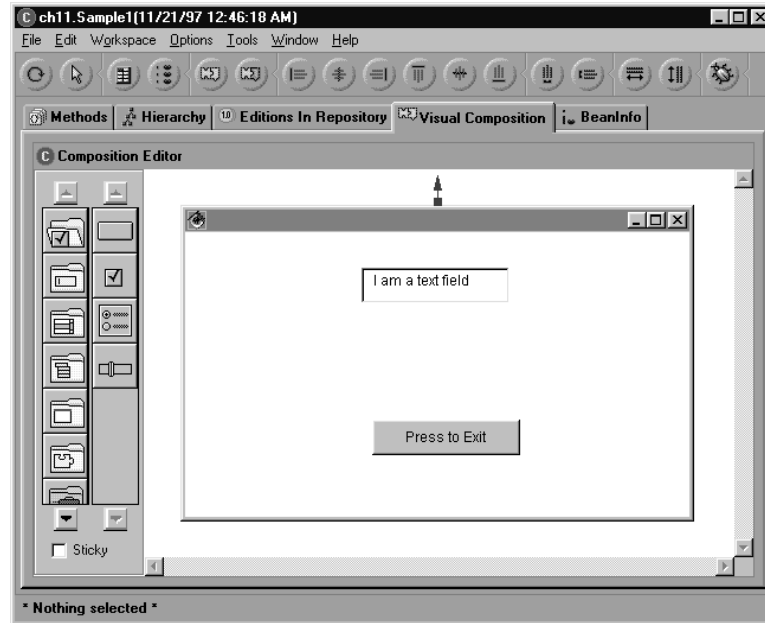


Figure 111. A Frame with a TextField and a Button.

To test your application, click the test icon of the toolbar (the first icon from the left). The Visual Composition Editor saves your bean and generates the Java code on the fly. Then, it displays the command line arguments dialog window and prompts you for arguments for your application. Because no arguments are needed, just click the **Run** button to proceed. The frame should be displayed with the text field and the button (Figure 112). Notice that pressing the button does not close the frame, but clicking the frame close icon does, thanks to the connection added by default.

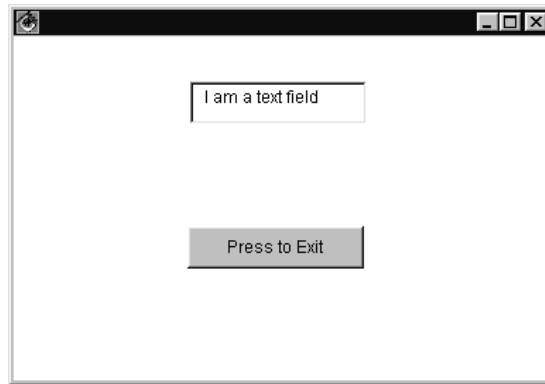


Figure 112. The Resulting Simple User Interface

Now let's have a look at the code that was generated for the simple application.

Looking at the Generated Code

You can access the code generated by the Visual Composition Editor by looking in the Methods or Hierarchy page in the Class Browser or by exporting the code to a file. The code shown below has been selected from the exported Java source file.

```
001 package chl1;
002
003 public class Sample1 extends java.awt.Frame implements
004     java.awt.event.WindowListener {
005     private java.awt.Button ivjButton1 = null;
006     private java.awt.TextField ivjTextField1 = null;
007
008     public Sample1() {
009         super();
010         initialize();
011     }
012
013     public Sample1(String title) {
014         super(title);
015     }
016
017     private void conn0(java.awt.event.WindowEvent arg1) {
018         try {
019             // user code begin {1}
020             // user code end
021             this.dispose();
022             // user code begin {2}
023             // user code end
024         } catch (java.lang.Throwable ivjExc) {
025             // user code begin {3}
026             // user code end
027             handleException(ivjExc);
028         }
029     }
030
031     private java.awt.Button getButton1() {
032         if (ivjButton1 == null) {
033             try {
034                 ivjButton1 = new java.awt.Button();
035                 ivjButton1.setName("Button1");
036                 ivjButton1.setBounds(155, 182, 125, 30);
037                 ivjButton1.setLabel("Press to Exit");
038                 // user code begin {1}
039                 // user code end
040             } catch (java.lang.Throwable ivjExc) {
041                 // user code begin {2}
042                 // user code end
043                 handleException(ivjExc);
044             }
045         };
046         return ivjButton1;
047     }
048     private java.awt.TextField getTextField1() {
049         if (ivjTextField1 == null) {
050             try {
051                 ivjTextField1 = new java.awt.TextField();
052                 ivjTextField1.setName("TextField1");
```



```

053     ivjTextField1.setText("I am a text field");
054     ivjTextField1.setBounds(161, 60, 125, 30);
055     // user code begin {1}
056     // user code end
057 } catch (java.lang.Throwable ivjExc) {
058     // user code begin {2}
059     // user code end
060     handleException(ivjExc);
061 }
062 };
063 return ivjTextField1;
064 }
065
066 private void handleException(Throwable exception) {
067     /* Uncomment the following lines to print uncaught exceptions
068        to stdout */
069     // System.out.println("----- UNCAUGHT EXCEPTION -----");
070     // exception.printStackTrace(System.out);
071 }
072
073 private void initConnections() {
074     // user code begin {1}
075     // user code end
076     this.addWindowListener(this);
077 }
078
079 private void initialize() {
080     // user code begin {1}
081     // user code end
082     setName("Sample1");
083     setName("Sample1");
084     setLayout(null);
085     setSize(426, 240);
086     add(getTextField1(), getTextField1().getName());
087     add(getButton1(), getButton1().getName());
088     initConnections();
089     // user code begin {2}
090     // user code end
091 }
092
093 public static void main(java.lang.String[] args) {
094     try {
095         chap11.Sample1 aSample1 = new chap11.Sample1();
096         try {
097             Class aCloserClass =
098                 Class.forName("uvm.abt.edit.WindowCloser");
099             Class parmTypes[] = { java.awt.Window.class };
100             Object parms[] = { aSample1 };
101             java.lang.reflect.Constructor aCtor =
102                 aCloserClass.getConstructor(parmTypes);
103             aCtor.newInstance(parms);
104         } catch (java.lang.Throwable exc) {};
105         aSample1.setVisible(true);

```

```
106 } catch (Throwable exception) {
107     System.err.println("Exception occurred in main() of
108                         java.awt.Frame");
109 }
110 }
111
112 public void windowActivated(java.awt.event.WindowEvent e) {
113     // user code begin {1}
114     // user code end
115     // user code begin {2}
116     // user code end
117 }
118 public void windowClosed(java.awt.event.WindowEvent e) {
119     // user code begin {1}
120     // user code end
121     // user code begin {2}
122     // user code end
123 }
124 public void windowClosing(java.awt.event.WindowEvent e) {
125     // user code begin {1}
126     // user code end
127     if ((e.getSource() == this) ) {
128         conn0(e);
129     }
130     // user code begin {2}
131     // user code end
132 }
133 public void windowDeactivated(java.awt.event.WindowEvent e) {
134     // user code begin {1}
135     // user code end
136     // user code begin {2}
137     // user code end
138 }
139 public void windowDeiconified(java.awt.event.WindowEvent e) {
140     // user code begin {1}
141     // user code end
142     // user code begin {2}
143     // user code end
144 }
145 public void windowIconified(java.awt.event.WindowEvent e) {
146     // user code begin {1}
147     // user code end
148     // user code begin {2}
149     // user code end
150 }
151 public void windowOpened(java.awt.event.WindowEvent e) {
152     // user code begin {1}
153     // user code end
154     // user code begin {2}
155     // user code end
156 }
157 }
```

Line 1: defines the `ch11` package where the `Sample1` class is defined.

Lines 3 through 6: define the `Sample1` class as a subclass of `Frame`. `Sample1` implements the `WindowListener` adapter to manage window events such as closing or resizing. The button and text field you add to the window are defined as private instance variables and follow the naming convention described in “The Free-Form Surface” on page 235.

Lines 8 through 11: define the `Sample1` constructor. First, the constructor of the base class is called, then an `initialize` method is called to initialize the button, text field, and connections.

Lines 17 through 29: define the `conn0` method, which is called by the `windowClosing` method to handle the closing event of the window. The `conn0` method calls the `dispose` method of the `Frame` class to clean up the memory.

Lines 31 through 47: define the accessor method for the button. Notice in line 32 that the accessor method creates the button the first time it is called by checking whether the `ivjButton1` instance variable is null.

Lines 48 through 64: define the accessor method for the text field. The text field is created the first time the method is called.

Lines 66 through 71: define a very useful method the lines of which can be uncommented to display any exceptions that could occur on the Console window.

Lines 73 through 77: define the `initConnection`, which adds the `Sample1` object (`this`) as a `WindowListener` to handle window events. Notice that `Sample1` is its own listener because `VisualAge` for Java does not support inner classes.

Lines 79 through 91: define the `initialize` instance method, which creates and adds the button and text field to the window and initializes the connections.

Lines 112 through 157: define the obstacle methods of the `WindowListener` interface as described in “Listener Types” on page 212.

As you can see, the code generated by the Visual Composition Editor is pure Java, just as you could write it yourself, provided you were using accessor methods for the fields in your classes.

Warning



We strongly recommend that you do not modify the code generated by the Visual Composition Editor. Indeed, when you regenerate the code, all of your modifications are overridden. However, as you can see in the generated code, locations are delimited by the `// user code begin` and the `// user code end` comments to allow users to add their own code. If code is added between these two markers, it will not be overridden during later regeneration.

More about JavaBeans

As you now know, the JavaBeans specification turns Java classes into reusable software components, called *beans*, that you can combine using a bean manipulator tool, such as the VisualAge for Java Visual Composition Editor, to create applets or applications. JavaBeans provide support for:

- ❑ **Portability:** Beans can be created and run on any Java platform.
- ❑ **Introspection:** The tool that you use to combine beans can automatically discover how a bean works.
- ❑ **Properties and customization:** Properties are a bean's attributes. They can have visual or nonvisual representation. A developer using a bean can customize the appearance and behavior of a bean by changing its properties.
- ❑ **Persistence:** The state of a bean can be saved and then reloaded.

In the sections that follow we detail the features of beans and explain how you can create your own visual and nonvisual beans.

Bean Features

The key to understanding and using a bean is to understand its features, that is, the **events**, **properties** and, **methods** that it **exposes**. We say that a bean exposes a feature by making it available to other beans. The features a bean exposes constitute its interface, and you can use only the features part of this interface to combine beans with each other, through a tool such as the Visual Composition Editor.

Events

Events are “fired” by a bean to indicate a condition or that a property has changed. For example, an employee bean could fire an event when his salary changes or when he is old enough to retire. Other beans can register their interest in these events and be notified of their occurrence. This event-sending mechanism is built on the knowledge of the event model that you gained in “Handling Events with Java AWT 1.1” on page 208.

Beans can both fire and listen for events defined in the Java 1.1 event model or any new event that follows the event model. Beans expose the events that they fire in their interface.

JavaBeans extends the new Java 1.1 event model with the addition of the property change event. This event allows beans to notify each other when specific properties change and to act on that information (see “Properties” on page 248). Figure 113 shows the implementation details of property change support.

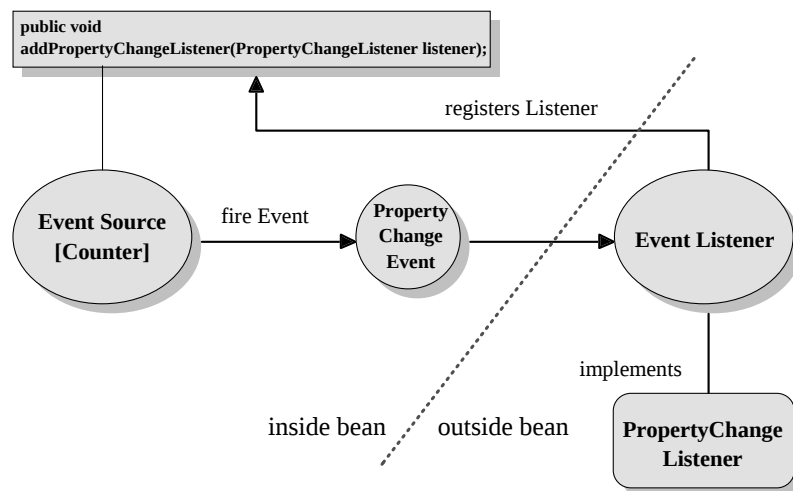


Figure 113. Property Change Support

Here is the sequence of calls or method invocations to support property change notification:

- ❑ The target bean or listener implements the `PropertyChangeListener` interface, that is, it implements the `propertyChange` method. This method contains the behavior to be executed when the event is fired.

- ❑ The target bean invokes the `addPropertyChangeListener` method on the event source bean (the bean that fires the property change).
- ❑ When the value of the property changes, the source bean fires a `PropertyChangeEvent` object to the listener, which in turns executes the `propertyChange` method.

Properties

The attributes exposed by a bean are called *properties*. Properties can be *read*, *written*, *bound*, and *constrained*. The properties exposed by a bean are typically a subset of its attributes.

The properties exposed by a bean are the data that it “knows” and whose value it wants other components to be able to get or set. The values are always accessed and manipulated through accessor and mutator methods that must follow a design pattern, which is in fact a simple naming convention defined by JavaBeans:

- ❑ For a property named *xxx*, you must create two methods, *getXxx* and *setXxx*, to access and change the value of the property. The type returned by the get method should be the same type as the argument passed to the set method.
- ❑ For a boolean property, you must define get and set methods by following the above convention. However, you can use the prefix “is” instead of “get.”

Properties can be categorized as *bound*, *indexed*, *constrained* or *editable*:

- ❑ **Bound properties** are properties that fire the property-change event when their value is changed. A `PropertyChangeEvent` includes the old and new values of the property. An example of a bound property would be the balance property of the `BankAccount`. The `setBalance` method would fire a property-change event so that the balance can be updated in a listener object.
- ❑ **Indexed properties** are simply properties that support a range of values. They are implemented using arrays and have four access signatures where the arrays of values can be accessed by either a single element or the entire array. For example, a list of transactions could be an indexed property of the `BankAccount`. The following methods would be defined to access the transactions:

- `void setTransaction(int index, Transaction aTransaction)`
 - `Transaction getTransaction(in index)`
 - `void setTransaction(Transaction[] transactions)`
 - `Transaction[] getTransaction()`
- ❑ **Constrained properties** are properties that can allow other beans to decide whether a change in the property's value is to be accepted. The other beans are notified through a `PropertyChangeEvent`, and they can throw a *PropertyVetoException* to veto the changes and to restore the properties to their old values.
- ❑ **Editable properties** are presented in the development environment so that the developer can customize the bean at development time. In VisualAge for Java you customize the bean through the Properties dialog. Programmers can provide their own custom property sheet for a particular bean. In this case the property sheet will be automatically invoked by the development tool when the bean is selected. A custom editor can also be created for a particular property of a bean. The ordinary property sheet is used, but the custom editor is invoked when the special property is edited.

Properties can also be designated as *hidden*, where they are visible only to the development environment, and *expert*, where they should be manipulated only by expert users.

Methods

Methods are the actions that a bean exposes for invocation by other components. The bean methods are a subset of the public methods of the class underlying the bean.

It is important to be aware of the naming convention used by JavaBeans when you write methods. Indeed, every accessor methods should follow the naming convention described in “Properties” on page 248 for each bean property, whereas other public methods are ordinary bean's methods and do not need to follow any specific naming convention.

Introspection and BeanInfo Class

The power of JavaBeans is evident when you use a JavaBeans-compliant application builder and drag a bean off its palette and drop it on a form. At this time, the application builder tool creates the bean by calling its constructor (which it can do if there is a default constructor defined) and extracts all the necessary information to create a property sheet and event handlers for the bean without accessing its source code.

The magic behind the scene lies in the *introspection* mechanism which allows an application builder to interrogate a bean to get its properties, methods, and events.

In short, JavaBeans proposes a standard way to the manipulation of beans by providing them with an “identity card.” Each bean can be associated with a *BeanInfo* class (the identity card), which gives all information regarding the bean. An *Introspector* class queries the bean by using a *getBeanInfo* method that returns a *BeanInfo* object from a *Class* handle parameter.

If a bean is not associated with a *BeanInfo* class, the application builder used a *reflection* mechanism to study the methods supported by the beans and then applies the simple naming convention defined by the JavaBeans specification (see “Properties” on page 248) to deduce from those methods which properties, events, and public methods are supported.

All Java classes are beans, because there is no separate specification language and no *BeanInfo* class is needed. However, many classes are not intended to be used as beans—they are intended as strictly programmatic interfaces. For example, many of the JDK classes are not intended to be used in visual development environments.

When we talk about beans in the remainder of this book, we are talking about Java classes that were designed to be used as beans, that is, they are intended to be used in a visual development environment: VisualAge for Java in our case.

Bean Persistence

As any Java class that implements the *Serializable* or *Externalizable* interface, a bean can be stored in a stream and restored later on. To store a group of beans, the JavaBeans specification describes a JAR file as a structures ZIP file containing multiple serialized objects, images, sounds, class files and so on (see also “Java Archive Files” on page 79). By compressing beans into one single file, beans can be downloaded in one piece from the Web, making applet downloading more efficient.

Using VisualAge to Create Beans

In “Creating a Simple User Interface” on page 238 you used several beans in VisualAge for Java because all AWT classes are beans. You have also unconsciously created beans whenever you created an application or an applet by combining primitive beans such as a button or a text field to build a more complex bean.

Before you implement your ATM machine in Chapter 12 and use the bank account classes you have already created in the previous chapters, we propose one more example to complete your training with the Visual Composition Editor.

In this example, you build a calculator that demonstrates how to use the BeanInfo page and how to take advantage of the multiple connection types available to wire your beans together. In addition, this example illustrates the model-view design we introduced at the beginning of this chapter.

Requirements for a Simple Calculator

Let’s build a very simple calculator that can add, subtract, multiply, and divide numbers of type double. To design our application using a model-view approach, we first create a calculator model that holds two numbers and a result and knows how to add, subtract, multiply, and divide two numbers.

Figure 114 shows a UML representation of the unique class of the calculator model, which you are going to implement as a nonvisual bean, using the BeanInfo page.

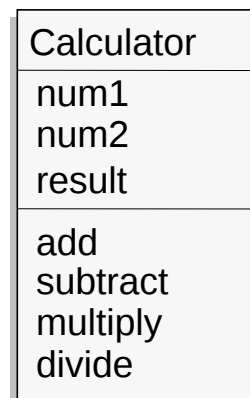


Figure 114. Calculator Model

In addition you need a view for the calculator that allows the user to enter two numbers, activate any of the four possible operations and see the result of the addition.

Figure 115 shows a simple calculator view that you are going to implement as a visual bean.

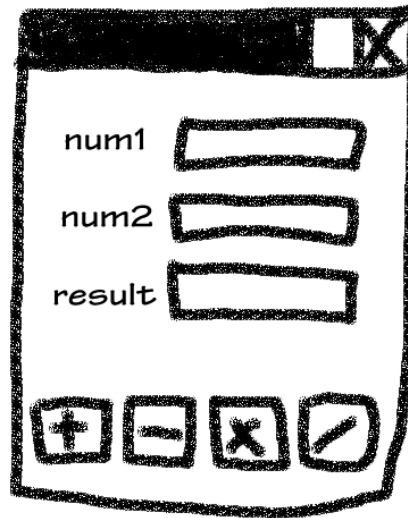


Figure 115. Calculator View

Building the Calculator Model

You use the BeanInfo page of the Class/Interface Browser to view, define, or modify bean interface features. When you add features in the BeanInfo page, you define the external view of your bean to consumers who use the bean. By contrast, when you compose a composite bean in the Visual Composition Editor, you define the internal content of the bean.

The BeanInfo Page

As you now know, the features of a bean interface consist of properties, events, and methods. These features represent the properties and behavior of your class.

The BeanInfo page (accessed from the BeanInfo tab in a class browser) is where you create and inspect the events, methods and properties of your bean. From the BeanInfo page you can also gen-

erate a BeanInfo class to explicitly describe your bean. In fact, VisualAge for Java generates a BeanInfo class for you if you modify the bean interface, by adding a property, for example.

From the BeanInfo page, you use the Features menu to manage bean features. You access the Features menu by either selecting it from the menu bar Features choice or opening it as pop-up menu from the Features pane.

When you add a new feature, VisualAge generates code for the feature and defines the extra methods you need for the different types of properties you may define.

Information



You may wonder whether you should create a Java class as a regular class by using the Create Class SmartGuide or as a bean by using the BeanInfo page. The choice depends on whether you intend to use your class in the Visual Composition Editor:

- ☐ If you know that you will combine your class with other beans by using the Visual Composition Editor, design your class as a bean from scratch by using the BeanInfo page.
- ☐ If you know that you will use your Java class to provide only a few interactions with beans and that you might not use it in the context of the Visual Composition Editor, create your class by using the Create Class and Create Field SmartGuides.

Building the Calculator Bean

Let's build the Calculator bean by using the BeanInfo page. First you create the Calculator class by using the Create Class SmartGuide. Then you switch to the BeanInfo page to add the properties and the methods.

Creating the Calculator Class

Using the Create Class SmartGuide, create a Calculator class which inherits from `java.lang.Object` in the `ch11` package of the Learn Java project. Make sure the **Write source code for the class** radio button and the **Browse the class when finished** checkbox are both selected before clicking the **Finish** button. The class browser opens with the Calculator class.

Adding Properties to the Calculator

To add the num1, num2 and result properties to the Calculator:

1. Switch to the BeanInfo page.
2. From the menu bar select **Features**→**New Property Feature...** to create the first bound property that will store the first number. The New Property Feature SmartGuide is displayed.
3. Enter *num1* in the Property name field, *double* in the Property type field. Make sure the **Readable** and **Writable** check boxes are selected and the **Indexed** checkbox is unselected. Select the **bound** radio button to make a bound property and click the **Finish** button to create the num1 property.
4. Repeat Step 2 and 3 for *num2* and *result*. Once the three properties have been added, the BeanInfo page should look like Figure 116.

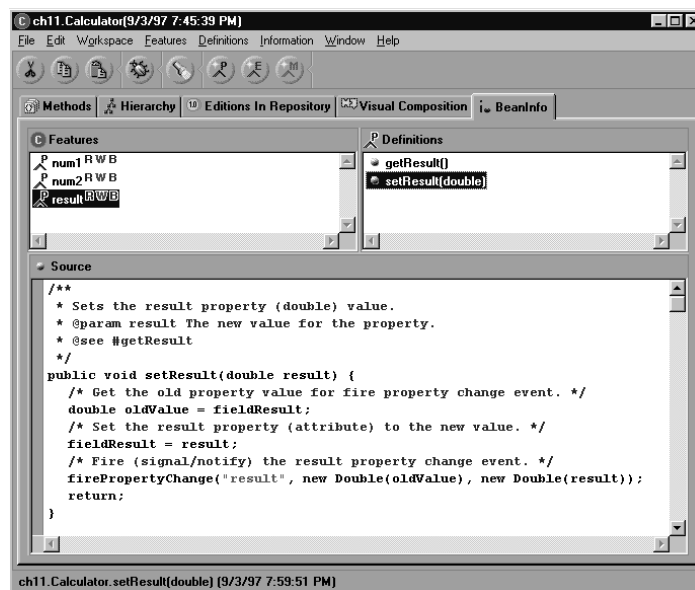


Figure 116. Calculator BeanInfo Page

Adding Methods to the Calculator

To add methods you proceed in the same way:

1. From the menu bar select **Features**→**New Method Feature...** to create the add method that will add num1 to num2. The New Method Feature SmartGuide is displayed.

2. Enter *add* in the Method name field and *void* in the Return type field. Make sure the Parameter count slider is set to 0. Then click the **Finish** button to create the add method.
3. Select the add method in the Definitions right pane to browse the source code in the bottom pane. Then modify the code as follows (in bold):

```
public void add() {  
    /* Perform the add method. */  
    setResult(getNum1() + getNum2());  
    return;  
}
```

4. Save the code by selecting the Save menu item from the Source pane pop-up menu.

To add the *subtract* method, repeat Steps 1 through 4 with the following code:

```
public void subtract() {  
    /* Perform the add method. */  
    setResult(getNum1() - getNum2());  
    return;  
}
```

To add the *multiply* method, repeat Steps 1 through 4 with the following code:

```
public void multiply() {  
    /* Perform the add method. */  
    setResult(getNum1() * getNum2());  
    return;  
}
```

To add the *divide* method, repeat Steps 1 through 4 with the following code:

```
public void divide() {  
    /* Perform the add method. */  
    setResult(getNum1() / getNum2());  
    return;  
}
```

If you edit an embedded bean in the Visual Composition Editor and then change its features in the BeanInfo page, be sure to refresh the bean interface when you return to the Visual Composition Editor. Selecting **Refresh Interface** from the bean pop-up menu.

Your calculator is ready to be hooked to the calculator view that you build in the next section.

Building the Calculator View

You build the calculator view by creating an application with the Visual Composition Editor. This application imbeds the Calculator bean that is wired to the calculator interface.

To create the `CalculApp`, follow these steps:

1. Use the Create Class SmartGuide to create a class. The class name is *CalculApp* and it should be created in the *ch11* package in the *Lean Java* project. The class derives from `java.awt.Frame`. Make sure the **Design the class visually** checkbox is selected before clicking the **Finish** button. The Visual Composition Editor opens with the `CalculApp` frame window inside.
2. Select the *Data Entry* category in the palette and click the **Sticky** checkbox at the bottom. Then select a *Label* bean from the palette and drop it three times inside the `CalculApp` client area. Notice that the Sticky checkbox allows you to drop multiple beans of the same type.
3. Double-click the first label to open its property sheet and change its text attribute to *num1* and its name to *Num1Label*. You can also change its font to your liking.
4. Repeat Step 3 for the two other labels but change their text property to *num2* and *result* and their name to *Num2TextField* and *ResultTextField*, respectively.
5. From the Data Entry category, select a *TextField* bean and drop it three times inside the `CalculApp`. Change the TextFields names to *Num1TextField*, *Num2TextField*, and *ResultTextField*, respectively. If needed, you can change the font property of each *TextField* by opening their property sheet.
6. Select the *Button* category from the palette and select the *Button* bean. Then drop four *Button* beans in the `CalculApp`.
7. Change the label property for each button respectively to **+**, **-**, **x**, and **/** and change the buttons names to *AddButton*, *SubtractButton*, *MultiplyButton*, *DivideButton*.
8. Adjust the buttons, text fields, and labels as shown in Figure 117.

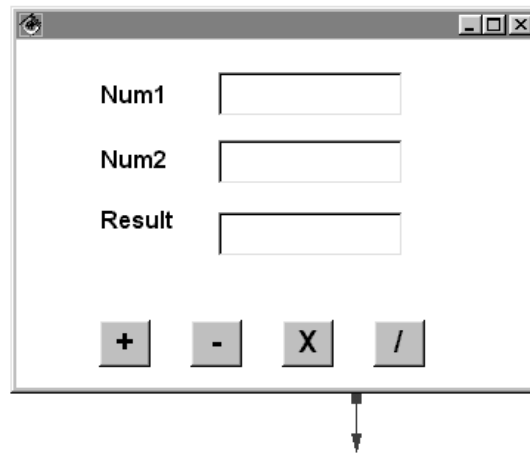


Figure 117. CalculApp Frame Window

Connecting the Model and the View

Now that you have created your calculator view and model, you just have to combine them to complete your calculator application.

You can combine visual and nonvisual beans any way you want as long as you combine them through their interface. You can only connect from the bean interface of a source bean to the bean interface of a target bean. You cannot connect to a hidden feature such as a private attribute which is not part of the bean's interface.

The Visual Composition Editor provides you with four main connections to wire your beans:

- ☐ Property-to-Property connections
- ☐ Event-to-Method connections
- ☐ Script connections
- ☐ Parameter connections

Let's describe each connection type by illustrating its use in the calculator application.

Property-to-Property Connections

A property-to-property connection links two property values together. This connection causes the value of one property to change when the value of the other changes, except as noted in Table 8.

A property-to-property connection appears as a bidirectional dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

When your bean is constructed at run time, the target property is set to the value of the source property. These connections never take parameters.

For indexed properties, VisualAge generates two get/set method pairs—one for the array and one for accessing elements within the array. When you connect indexed properties, VisualAge uses the accessors for the entire array. If you want to access an individual element, make a method connection to the specific accessor.

To achieve the behavior that you anticipate, you must know something about the properties you are connecting. Table 8 shows the results of connecting properties of different types depending on whether or not the property change is associated with an event and a set method is defined for setting the property.

Table 8. Property-to-Property Connection Results

	And the target has both a set method and event	And the target has an event but no set method	And the target has a set method but no event
If the source has both a set method and event...	Source and target values are fully synchronized.	VisualAge automatically reverses the connection.	The source initializes the target. The target is updated whenever the source's value changes.
If the source has an event but no set method...	The source initializes the target. The target is updated whenever the source's value changes.	This connection is not valid.	The source initializes the target. The target is updated whenever the source's value changes.
If the source has a set method but no event...	The source initializes the target only. The source is updated whenever the target's value changes.	VisualAge automatically reverses the connection.	The source initializes the target. No further updates occur.

Because most of the properties in this version of AWT are not bound, the source initializes the target when constructed and performs no further updates (as stated in Table 8). If you want to use the properties in an event-to-method connection, you must associate an event with the property.

If at least one property is constrained and a change is vetoed, the properties are not synchronized.

If properties are not synchronized, it may be because the event to which they are bound is asynchronous. In this case you should refer to the bean information for details on selecting another event.

Before drawing any connection in your *CalculApp* example, you must add the Calculator bean to the free-form surface of *CalculApp*:

1. Select **Option**→**Add Bean...** from the menu bar to display the Add Bean dialog window.
2. Type in *Calculator* in the class name entry field and click the **Browse** button.
3. Select *Calculator* in the Class names list and verify that *ch11* is displayed in the Package names list.
4. Click **OK** to validate your choice.
5. Complete the Add Bean dialog window with the name you want to associate with your Calculator bean (*Calculator* for example) and click the **OK** button to load the cursor with the Calculator bean.
6. Drop the Calculator bean anywhere on the free-form surface. Notice that you cannot drop it on the *CalculApp* frame.

For more information about adding beans to the free-form surface, refer to “Adding a Bean by Specifying Its Class Name” on page 236.

You can verify that the Calculator bean you dropped on the free-form surface is a real living object by opening its property sheet (Figure 118). Indeed, when you drop the Calculator on the free-form surface, its default constructor is called, and the properties of type double are initialized to 0.0 by Java.

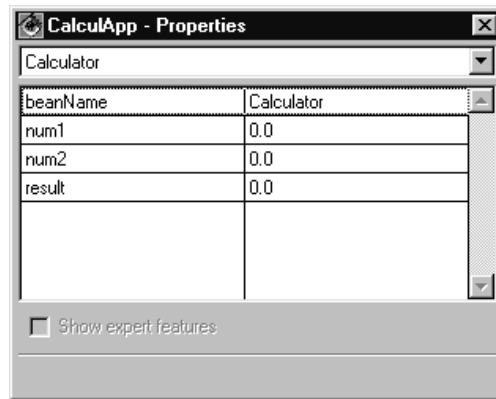


Figure 118. Calculator Property Sheet

Now that you have added the Calculator bean to the free-form surface, you need to use property-to-property connections to link the text attribute of the TextField beans to the num1, num2, and result properties of the Calculator bean.

To link the text property of the first TextField bean to the num1 property of Calculator, follow these step-by-step instructions:

1. Select the Num1TextField bean and activate its pop-up menu by clicking the right mouse button.
2. Slide the mouse down to the Connect option; a submenu is displayed. Choose the Connect the text property from the submenu. A dotted line with a spider at its end appears.
3. Drag the spider to the Calculator bean and click the left mouse button. The pop-up menu of the Calculator is displayed to let you choose a property.
4. Choose the num1 property to complete the connection.
5. Double-click the connection you just created and replace *<null>* in the Source event combo box with *text* (Figure 119). This setting ensures that whenever the content of Num1TextField changes, the property-to-property connection is fired and the calculator num1 field is updated accordingly.

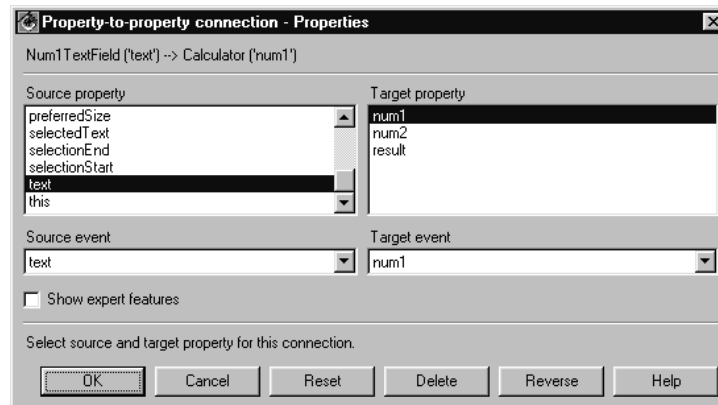


Figure 119. Selecting a Source Event for a Property-to-Property Connection

Repeat Steps 1 through 4 for the remaining TextField beans and connect their text property to the num2 and result properties of Calculator bean (Figure 120). Notice that you must draw the last connection between ResultTextField and the result property from the result property to ResultTextFied. This ensures that the result is propagated from the model to the view.

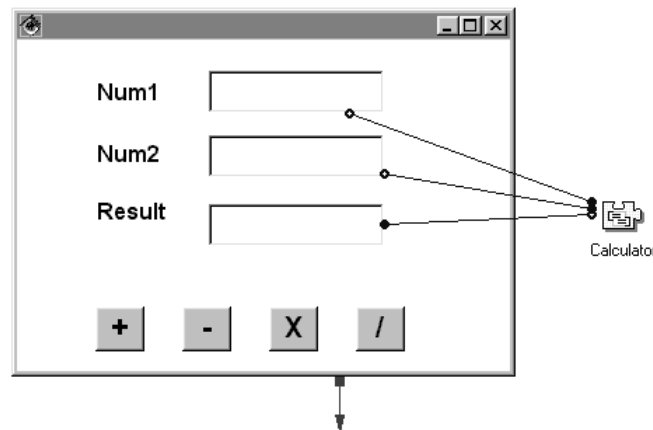


Figure 120. Property-to-Property Connections of the CalculApp

Event-to-Method Connections

An event-to-method connection calls the specified method of the target object whenever the source event occurs. Often a good deal of the behavior of an application can be specified visually by causing a method of one bean to be invoked whenever an event is signaled by another bean.

An event-to-method connection appears as a unidirectional dark green arrow with the arrowhead pointing to the target.

To complete your Calculator application, you must add four event-to-method connections to fire each arithmetic operation when the associated button is clicked by the user.

To add the first event-to-method connection:

1. Select the AddButton bean and activate its pop-up menu by clicking the right mouse button.
2. Choose the *actionPerformed(java.awt.event.ActionEvent)* event from the pop-up menu. A dotted line with a spider at its end appears.
3. Drag the spider to the Calculator bean and click the left mouse button. The pop-up menu of the Calculator is displayed to let you choose a method.
4. Choose **All Features...** from the pop-up menu to display the list of all the features exposed by the bean (Figure 121).
5. Select the `add()` method from the Method list and click the **OK** button to validate.

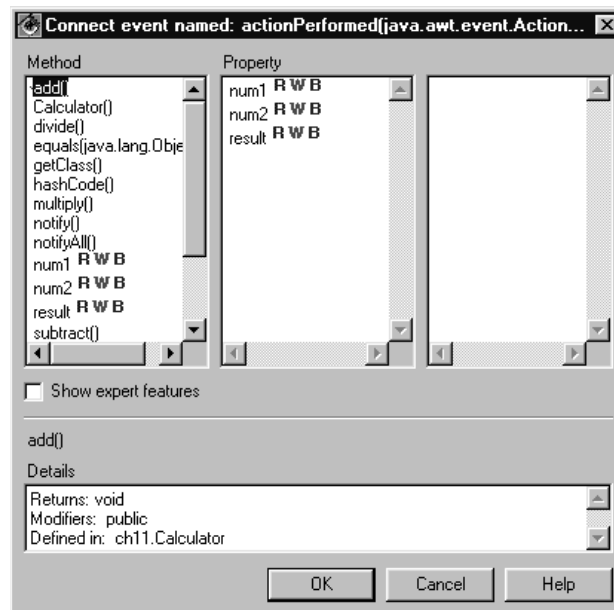


Figure 121. Calculator Bean Method and Property Features

Only the properties and methods are listed in Figure 121 because you can connect an event only to one of these two entities.

Build the three remaining event-to-method connections for the other buttons to activate the subtract, multiply, and divide methods of the Calculator bean. Once all your connections are drawn, your application should look like that in Figure 122.

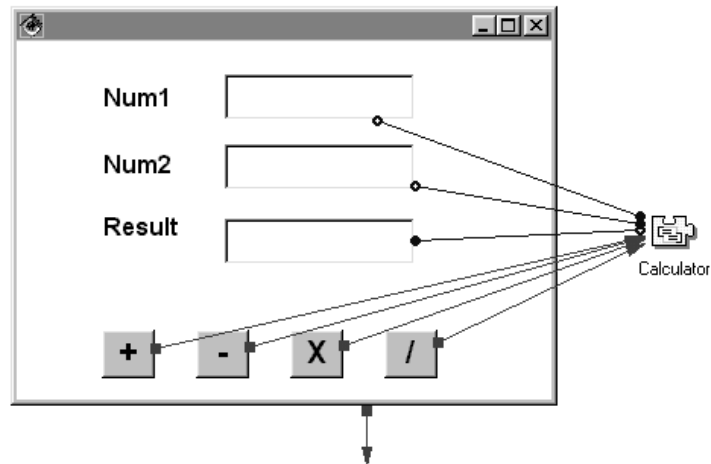


Figure 122. CalculApp with All Connections

Because the methods of the Calculator bean do not require any parameter to execute, the event-to-method connections you draw are plain. Sometimes methods need parameters to execute. In that case, an event-to-method connection appears dotted until all parameters are provided to execute the method.

You have two ways of providing parameters: by editing the connection properties or by using parameter connections as described in “Parameter Connections” on page 266.

You can now test your application by clicking the **Test** button of the Visual Composition Editor. Notice that the result text field is already initialized with 0.0. In effect, when running the application, the Calculator bean default constructor is called and the properties of type double are initialized to 0.0 by Java. This value is used to initialize the text attribute of the ResultTextField bean through the corresponding property-to-property connection.

Script Connections

To access behavior that is not part of the bean interface, you can use script connections.

It often happens that you want some processing to occur when an event is signaled by one of the beans in the composition you are editing, but none of the beans in the composition publishes a public beans method that does exactly what you want. In this case, VisualAge enables you to write a custom private method in the class you are editing and then connect to that method. We call these custom private methods of the class being edited *scripts*, to distinguish them from the public methods that you may have created for your composite class and published as a bean method. In fact, there is no requirement that scripts be private, and they are not really different from other Java methods; *scripts* are only really scripts by virtue of the way you intend to use them.

This may sound a little confusing, but the problem is that in Java, the word *method* is used in two subtly different ways. In the Java language, a method refers to any method of any class. However, in JavaBeans, method refers to a subset of the Java methods of a class that are exported as features of the bean. The set of JavaBeans method features of a bean is often the same as the set of Java public methods of the bean, but the bean provider may further restrict the set of Java methods that show up as JavaBeans features.

Note that there is no way to connect to private methods of the beans in your composition. You can only connect to private and public methods of the class you are editing, or public methods of any of its embedded beans.

This type of connection appears as a unidirectional dark green arrow with the arrowhead pointing to the free-form surface.

Examples of potential scripts are:

- ☐ Local methods declared as private or protected (or those defaulting to package scope: no access modifier)
- ☐ Inherited methods declared as protected

Connect to scripts instead of to methods when you want to keep the operation internal to the class. For example, suppose that you want the `ResultTextField` bean of the `CalculApp` to display its value with a different color depending on whether the result of an operation is negative. To change the foreground color property of `ResultTextField`, you need to do a little manipulation of its text property to convert it to a double and test whether it is negative or positive. If users of the composite application do not need to be aware that this processing is happening, you can use a script.

Let's create an `adjustColor` script and its associated script connection:

1. Select `ResultTextField` and activate and choose the `textValueChanged(java.awt.event.TextEvent)` event. A dotted connection with a spider at its end is displayed.
2. Drag and drop the spider on the free-form surface that represents the `CalculApp` bean and select **Event to script...** from its pop-up menu. A script dialog window is displayed (Figure 123). The **instance** button is a switch that allows you to select from instance to class methods. The **New method...** button displays the New Method SmartGuide to let you create a new method.
3. Click the **New method...** button and create a private method with the following signature: `private void adjustColor()`. Validate the creation by clicking the **Finish** button. The `adjustColor` method is added to the list of methods and is highlighted (Figure 123).
4. Click the **OK** button to validate the choice of `adjustColor` as a script and close the script dialog window.
5. Switch to the Method page of `CalculApp` and select the **adjustColor** method you just created. Modify the code as follows (in bold):

```
public void adjustColor() {
    if ((new Double(getResultTextField().getText())).
        doubleValue() < 0.)
        getResultTextField().setForeground(Color.red);
    else
        getResultTextField().setForeground(Color.black);
    return;
}
```

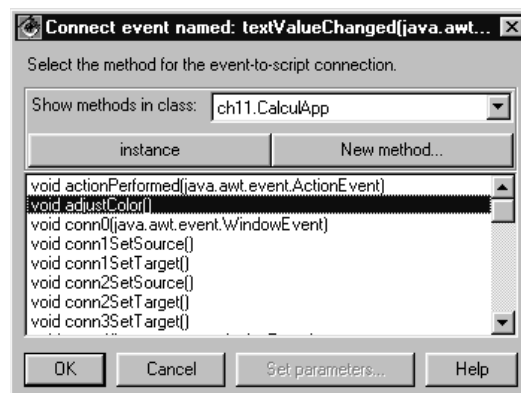


Figure 123. Script Dialog Window

You can now test your new application by entering different values and checking that any negative value is displayed in red.

Notice in the code of the `adjustColor` method that you access the result field by using the default accessor generated by the Visual Composition Editor. Notice also that the black and red colors are obtained directly as class variables from the `java.awt.Color` class.

Parameter Connections

A parameter connection supplies an input value to the target of a connection by passing either a property's value or the return value from a method. In a parameter-to-method connection, the connection appears as a unidirectional violet arrow with the arrowhead pointing from the parameter of the original connection to the method providing the value. In a parameter-to-property connection, the connection appears as a bidirectional violet line with the dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. The original connection is always the source of a parameter connection; the source feature is the parameter itself.

If you select the parameter as the target, VisualAge reverses the direction of the parameter connection automatically. If the target of the original connection takes parameters and the same event provides parameters by default, the connection line might appear solid. This is true even if the target takes one input parameter and you have not otherwise provided one. VisualAge can use any of the following means to supply parameters with values:

- ❑ If the parameter is connected to a property, the connection calls the property's `get` method to get the property's value and return it to the parameter.
- ❑ If the parameter is connected to a method, the connection code calls the method and passes the method's return value to the parameter.
- ❑ If the source of the original connection passes event data in the connection code, VisualAge applies it to the parameter. If several values are required, event data is applied to the first parameter only. If you specify a constant parameter value in the original connection, VisualAge passes it in the connection code.

Let's suppose now that you want to have a more fancy calculator application by letting the user select the color. A `ColorEditor` bean is available in VisualAge for Java from the `sun.beans.editors` package, which provides different editors for primitive data types and `Color` objects. You use this editor to allow the user to pick a color for the result text field as follows:

1. Select **Options**→**Add Bean...** to add the ColorEditor bean in the application panel (Figure 124). You may have to resize the editor in the free-form surface first before dragging and dropping it in the application panel.

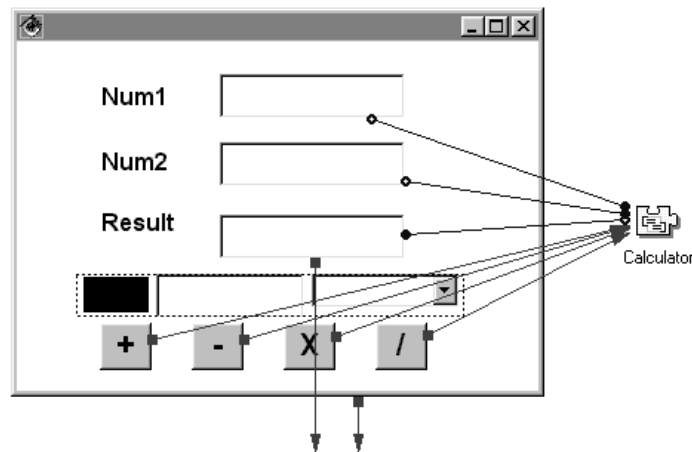


Figure 124. CalculApp with the ColorEditor

2. Switch to the Method page and select the `adjustColor` method. Modify the method by supplying a parameter for the color as follows:

```
public void adjustColor(Color aColor) {
    if ((new Double(getTextField3().getText())).
        doubleValue() < 0.)
        getTextField3().setForeground(aColor);
    else
        getTextField3().setForeground(Color.black);
    return;
}
```

3. Switch back to the Visual Composition Editor and double-click the event-to-script. The `adjustColor()` method is no longer selected, and you must reattach the connection to `adjustColor(Color aColor)` instead.
4. Provide the color selected by the user by connecting the `value` property of the ColorEditor bean to the `aColor` parameter of the event-to-script connection. You just created a parameter connection to supply the color parameter to the `adjustColor` method (Figure 125).

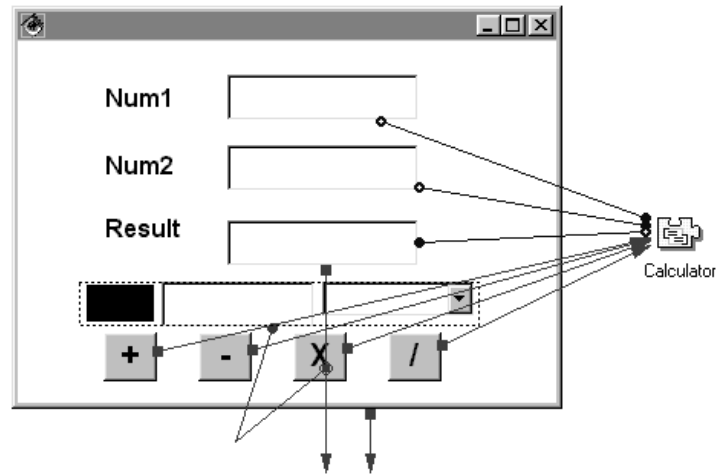


Figure 125. Final Calculator Application

One last time, you can generate your code and test the application by picking a color from the color editor and verify the impact in the result field.

Notice that if you change the color from the color editor, the color of the result text field does not change because there is no direct connection between the color editor and the text field.

Try this now:

1. Select the red color with the color editor of the calculator.
2. Subtract 1 from 3 with the calculator (a red-colored -2 is displayed in ResultTextField).
3. Change the color set in the color editor to green and press the subtract button again. The color of ResultTextField is not updated!

You may find it a little bit puzzling that, when you pick a different color and issue again the same subtraction that results in a negative number, the color of the result text field does not change. The reason is tricky. In fact, when you press the subtract button to recalculate the result, the result value of the Calculator bean does not change, and the setResult method does not send any event. For this reason the property-to-property connection from the result property to the text property of the result text field is not fired, and the text property does not change either. Finally, because the text property of the result text field does not change, an event is not generated, and the event-to-script connection is not fired. Hence, the color of the result text field is not updated. To

update the color, you would need to create an event-to-method connection from the *actionPerformed(java.awt.event.ActionEvent)* of the subtract button to the *dispatchEvent* method of *ResultTextField* and provide an event parameter to the connection by entering in its property sheet:

```
new java.awt.event.TextEvent(this,  
    java.awt.event.TextEvent.TEXT_VALUE_CHANGED)
```

The *dispatchEvent* method allows the result text field to send a *textValueChanged* event as if its text property had changed. Notice that you need to draw the same connection for the other buttons because the other operations can also produce a negative result when either *num1* or *num2* is negative.

Additional Connections

Two other connections are available to connect beans but you will not use them as often as you use the four main connections:

Property-to-Method Connections

A property-to-method connection calls a method whenever the source property is bound and changes value. This connection is similar to an event-to-method connection because the connection calls the method when the property's event is signaled. A property-to-method connection appears as a unidirectional dark green arrow with the arrowhead pointing to the target. The property's value is passed as the first parameter of the method if a parameter is not explicitly specified. If the method requires more than one input parameter, the connection line initially appears dashed to show that it is incomplete. To make the connection complete, you must provide the input parameter, either through a parameter connection or by setting a constant parameter value.

Event-to-Property Connections

An event-to-property connection updates the target property whenever the source event occurs. An event-to-property connection appears as a unidirectional dark green arrow with the arrowhead pointing to the target. The property must have a public set method known to VisualAge; otherwise, you cannot make the connection. If you open properties on an event-to-property connection, the target of the connection appears as a method with the same name as that of the target property.

Adding Menus to Your Application

In “Creating and Using Menus” on page 224, you learned how to program menus for your applications or applets. In this section we show you how to use the Visual Composition Editor to create menus visually.

Menus and Menu Bars

As you know from Chapter 10, menus can exist only in menu bars, and menu bars can be attached only to windows (specifically, to frames). For this reason, you cannot attach menus to an applet because applets do not have frames to which to attach the menu bar. However, you still can use pop-up menus with applets.

Let’s provide a menu bar and two menus to *CalculApp*: *File* and *Operation*. The *File* menu provides a *Reset* option to reset the calculator fields to 0.0 and an *Exit* option to exit from the application. The *Operation* menu provides an alternative option to execute the four operations using menu items Add, Subtract, Multiply, and Divide.

Follow these step-by-step instructions to provide menus for your application:

1. Open the *ch11.CalculApp* class in the Visual Composition Editor.
2. Select the **Menus** category.
3. Select a **MenuBar** bean from the palette (first bean in the palette).
4. Drop the **MenuBar** bean on the *CalculApp* frame. Notice that you cannot drop it elsewhere. A **MenuBar** bean can only be attached to a window frame. When you drop the **MenuBar** bean, a **Menu** bean is added automatically to the free-form surface and associated with the menu bar.
5. Double-click the **Menu** bean and change its label to *File*.
6. Select the **Menu** bean from the palette (last bean in the palette) and drop it in the menu bar. Notice that you cannot drop it elsewhere.
7. Double-click the **Menu** bean you just added and change its label to *Operation*.
8. Select a **MenuItem** bean from the palette (second bean in the palette) and drop it in the *File Menu* bean. Notice that you can drop a **Menu** bean inside a **Menu** bean to create cascading menus.

9. Double-click the MenuItem bean and change its label to Reset and add Ctrl+R as a shortcut.
10. Select a **MenuSeparator** bean from the palette (third bean in the palette) and drop it in the File Menu bean after the Clear MenuItem.
11. Select another **MenuItem** bean from the palette and drop it in the File Menu bean.
12. Double-click the MenuItem bean and change its label to Exit and add Ctrl+X as a shortcut.
13. Check the **Sticky** checkbox.
14. Select the MenuItem bean from the palette and drop four beans in the Operation Menu bean.
15. Uncheck the **Sticky** checkbox.
16. Change their labels to Add, Subtract, Multiply, and Divide. You can add shortcuts to them if you want.
17. Connect the *action* event from the Reset menu item to the *num1* field of the Calculator bean.
18. Double-click the connection and click the **Set parameters...** button to provide 0.0 as a parameter.
19. Repeat Steps 6 and 7 for the remaining fields.
20. Connect the *action* event from the Exit menu item to the *dispose* method of the CalculApp.
21. Connect the *action* event from the Add menu item to the *add* method of the Calculator bean.
22. Connect the *action* event from the Subtract menu item to the *subtract* method of the Calculator bean.
23. Connect the *action* event from the Multiply menu item to the *multiply* method of the Calculator bean.
24. Connect the *action* event from the Divide menu item to the *divide* method of the Calculator bean.

Once you have added all the connections, your application should look like that in Figure 126.

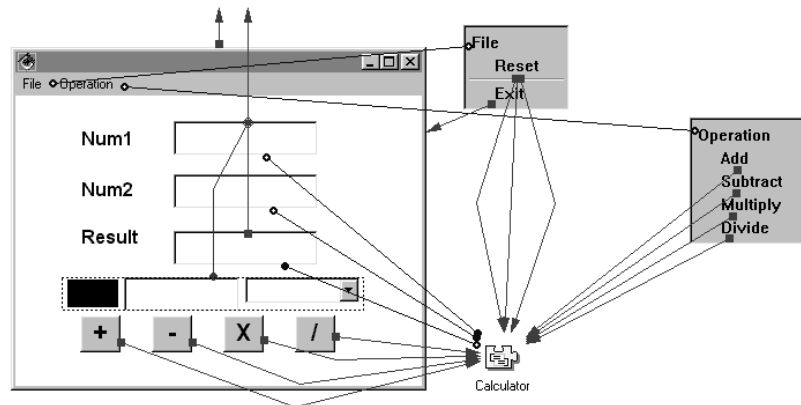


Figure 126. CalculApp with Menu Bar

Save **CalculApp** and run it. Try the different menu items and verify that the menu shortcuts work as expected.

Tip



You may experience difficulty setting the parameter of a connection if the parameter is already set to an initial value. In this case change the value of the parameter to any value and validate the change. Then change it back to the original value.

Pop-up Menus

Pop-up menus can be attached to any control. In particular, pop-up menus can be associated with applets. In the following instructions, you add a pop-up menu as an alternative to the **Operation** menu:

1. Open the **ch11.CalculApp** class in the Visual Composition Editor.
2. Select the **Menus** category.
3. Select a **PopupMenu** bean from the palette (fourth bean in the palette).
4. Drop the **PopupMenu** bean on the free-form surface. Notice that you cannot drop it elsewhere.
5. Double-click the **PopupMenu** bean and change its label to **OpPopupMenu**.
6. Select the **MenuItem** bean from the palette and drop it in the menu bar. Notice that you cannot drop it elsewhere.

7. Check the **Sticky** checkbox.
8. Select the **MenuItem** bean from the palette and drop four beans in the **OpPopupMenu** bean.
9. Uncheck the **Sticky** checkbox.
10. Change the labels of the **MenuItem** beans to Add, Subtract, Multiply, and Divide.
11. Connect the action event from the Exit menu item to the *dispose* method of the **CalculApp**.
12. Connect the *action* event from the Add menu item to the *add* method of the **Calculator** bean.
13. Connect the *action* event from the Subtract menu item to the *subtract* method of the **Calculator** bean.
14. Connect the *action* event from the Multiply menu item to the *multiply* method of the **Calculator** bean.
15. Connect the *action* event from the Divide menu item to the *divide* method of the **Calculator** bean.
16. Connect the *windowOpened* event from the **CalculApp** to the *add(java.awt.PopupMenu)* method of **CalculApp**.
17. Make a parameter connection from the *this* property of the **OpPopupMenu** to the previous connection. This connection passes the *OpPopupMenu* handle to the *add* method to associate the popup menu with **CalculApp**.
18. To activate the popup menu, connect the *MouseClicked* event from the **CalculApp** frame to the *show(java.awt.Component,int,int)* method of **OpPopupMenu**.
19. Double-click the previous connection and provide 50 and 50 for the x and y coordinates.
20. Connect the *this* property of the **CalculApp** to the *origin* parameter of the connection made in Step 18 (Figure 127).
21. Save and test your application.

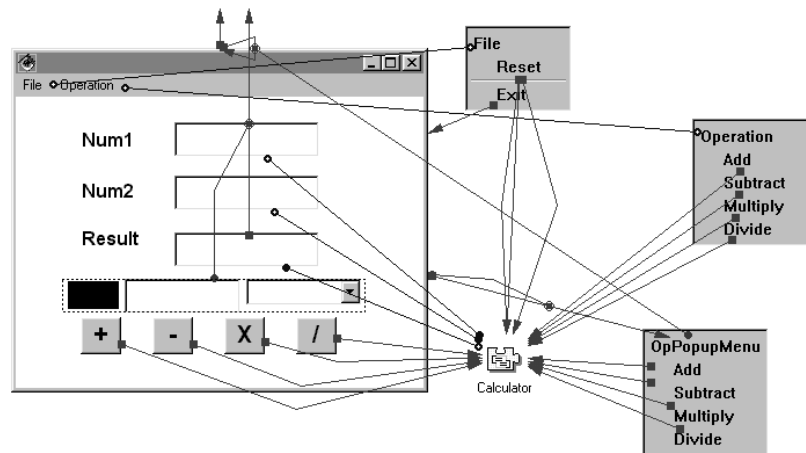


Figure 127. CalculApp with Menu Bar and Pop-up Menu

Notice that the pop-up menu is always displayed at the same location because you set the coordinates to 50 and 50 in Step 19. You can change the connection drawn in Step 18 to a script connection and use the `getX()` and `getY()` method from the `MouseEvent` object to display the menu where the mouse has been clicked.

Applets can be provided as well with pop-up menus. In this case create an event-to-method connection from the applet *init* event to the applet `add(java.awt.PopupMenu)` method to add a popup menu to your applet when it is initialized as described in Step 16.

Importing and *Beanizing* Existing Java Code

If you have existing Java code, you can easily import it into the VisualAge for Java environment. If the code is based on Java 1.0, you may want to make the transition to 1.1.

If you want to make your existing code behave as a bean, you can import it into the IDE, and then you have several choices:

- ☐ Adaptor classes: Create a lightweight class that provides an interface between other JavaBeans and your existing classes.
- ☐ Beanize your existing code: Manually add the event handling code to your class and then create a `BeanInfo` class that exposes the current properties and actions in the code.
- ☐ Delete and then re-create your properties within the `BeanInfo` page.

Your decision depends on the amount and complexity of the legacy code; the effort in re-creating properties; the effort in beanizing your existing code.

One important thing to realize is that the IDE will be introspecting your code. Your initial code is important here, for example, whether the code has getters and setters on properties.

In Chapter 12 we show you how to modify your bank model and build up beans from the classes you have already developed to connect them easily to the ATM interface you create.

Summary



Beans are software components that comply with the JavaBeans specification and can be visually manipulated in an application builder such as the Visual Composition Editor.

Beans expose events, methods, and properties that you can connect with other bean features by using different types of connections to visually create applications or applets.

12

Visual Programming in Action

In Chapter 11 you discovered how easy it is to visually program your applets or applications by using the Visual Composition Editor. In this chapter you continue to learn about the Visual Composition Editor by implementing the ATM applet.

In the first part of this chapter you expand the banking model by adding a few classes that are needed for your applet. You also modify some Java classes you built in the first part of the book to support the JavaBeans specification and ease the integration of the ATM view and the banking model into the Visual Composition Editor.

In the second part of this chapter, you build the view of your ATM applet. You discover how easy it is to use layout managers with the Visual Composition Editor and how you can create and reuse your own beans.

In the last part of the chapter, you put your applet together by connecting your beans with the connections you learned in Chapter 11.

Expanding the Banking Model

In the first part of this book you built a solid model for your ATM machine. However, you still need some classes to conduct a complete transaction. For example, users have to identify themselves to the ATM machine by using an ATM card of some sort. In addition, the ATM machine is meant to be used by several users with different accounts. Therefore a Bank class is needed to hold the list of all accounts users can interact with, through the ATM, assuming they have the proper ATM card to access them.

Warning



Before starting modifying the Banking Model and build the ATM user interface, you must ensure that the additional IBM beans included on the CDROM that accompanies this book are loaded in your workspace. Refer to Appendix A, “VisualAge for Java Installation,” on page 441 for more information about loading those beans in your workspace.

Creating an AtmCard Class

A user uses an ATM card to access his or her account from the ATM machine. To make it simple, we assume that a user has only one account and that the ATM card is associated with this account by an aggregation relationship. The ATM card has a card number and a personal identification number (PIN). The PIN allows the user to identify himself or herself on the ATM machine, and the ATM card can check the PIN entered by the user on the ATM machine against the PIN.

Figure 128 shows the UML representation of the AtmCard class with its fields and method.

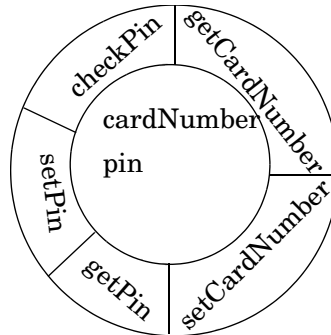


Figure 128. AtmCard Class

Because the AtmCard class will not interact directly with the ATM view, you will not build it as a bean. Rather, you will create it as a regular Java class.

Once again, you need to copy your previous classes from *ch08* into the new package for the chapter. Select the *ch08* package in the Workbench, select **Selected**→**Copy**, and click **Finish**. Enter *ch12* in the Copy dialog that appears and click **OK**.

You build the AtmCard class in three phases: first you use the Create Class SmartGuide to create the class, then you add the fields, and finally you add the methods.

Creating the AtmCard Class

To build the AtmCard class, follow these steps:

1. Start the Create Class SmartGuide to create your AtmCard class.
2. Fill in the Class Name entry field with *AtmCard* and check the **Browse the class when finished** checkbox. Then click the **Next>** button twice.
3. Click the **Add...** button to specify an interface to implement and choose: *java.io.Serializable*.
4. Click the **Finish>** button to create the AtmCard class and open the Class Browser window.

Adding Fields

Now that you have created your class, you can add fields from the Class Browser window:

1. Switch to the **Methods** page. From the **Methods** pull-down menu, select **New Field...** and add to the `AtmCard` class a *pin* field of type `String` with a *private* access modifier.
2. Add a *cardNumber* field of type `String` with a *private* access modifier.

Adding Methods

Finally you can add methods by following these steps:

1. Select **New Method...** from the **Methods** pull-down menu and add the private method, *void setCardNumber(String aCardNumber)*, to set the card number.
2. Modify the *setCardNumber* method:

```
private void setCardNumber(String aCardNumber) {  
    cardNumber = aCardNumber;  
    return;  
}
```

3. Add a public method, *String getCardNumber()*, to get the card number of the ATM card.
4. Modify the *getCardNumber* method:

```
public String getCardNumber() {  
    return cardNumber;  
}
```

5. Add a private method, *void setPin(String aPin)*, to set the PIN of the ATM card.
6. Modify the *setPinNumber* method as follows:

```
private void setPin(String aPin) {  
    pin = aPin;  
    return;  
}
```

7. Add a private method, *String getPin()*, to get the PIN of an ATM card.
8. Modify the *getPin* method:

```
private String getPin() {
    return pin;
}
```

9. Add a public method, *boolean checkPin(String aPin)*, to check the PIN of an ATM card.
10. Modify the checkPin method:

```
public boolean checkPin(String aPin) {
    return pin.equals(aPin);
}
```

11. Add the following constructors to the AtmCard class:

```
AtmCard(String aCardNumber, String aPin) {
    setCardNumber(aCardNumber);
    setPin(aPin);
}

AtmCard(String aCardNumber) {
    this(aCardNumber, "1234");
}
```

You have now created the AtmCard class.

Modifying the BankAccount Class

In this section, you associate the AtmCard class with the BankAccount abstract class. But most important, you need to transform the BankAccount fields and methods into bean features to make the BankAccount easier to use with the Visual Composition Editor. We call the process of changing an already existing Java class to a bean, *beanizing the Java class*.

Supporting the AtmCard Class

To aggregate the AtmCard class with the BankAccount class, you create a new private field and the associated accessor methods and a new constructor:

1. Select the BankAccount class in the ch12 package and open the Class browser.
2. From the Methods pull-down menu, select **New Field...** and add the *atmCard* field of type *AtmCard* with a *private* access modifier. Because this field will never interact directly with other bean objects in the Visual Composition Editor, you do not have to implement it as a property feature.

3. From the Methods pull-down menu, select **New Method...** and add the public method, *void setAtmCard(AtmCard aCard)*, to set the ATM card.

4. Modify the setAtmCard method:

```
private void setAtmCard(AtmCard aCard) {  
    atmCard = aCard;  
    return;  
}
```

5. Add a public method, *AtmCard getAtmCard()*, to retrieve the ATM card.

6. Modify the getAtmCard method:

```
public AtmCard getAtmCard() {  
    return atmCard;  
}
```

Now you have to modify the BankAccount constructor to set the properties of an associated AtmCard when a BankAccount object is created:

1. Create a new Method Feature, using the pull-down menu **Features→New Method Feature...**
2. Select *Constructor* from the **Create a new** drop-down list box and enter *BankAccount(String anAccountId, String aCardNumber)* as the constructor name. Click **Finish** to create the method feature.
3. Modify the code of the constructor:

```
public BankAccount(String anAccountId, String aCardNumber) {  
    setAccountId(anAccountId);  
    setLog(new Vector());  
    setAtmCard(new AtmCard(aCardNumber,  
        aCardNumber.substring(0, 2) + "123"));  
}
```

For simplification, the ATM card PIN is calculated from the ATM card number. This does not prevent the user from having multiple accounts for the same card.

The AtmCard class must also be associated with the CheckingAccount and SavingsAccount classes that you will use for the ATM application. Therefore, you must implement overloaded constructors for these two classes:

1. Select the CheckingAccount classes from the chap12 package in the Workbench.

2. Create a new Method Feature, using the pull-down menu **Features**→**New Method Feature...**
3. Select *Constructor* from the **Create a new** drop-down list box and enter *CheckingAccount(String anAccountId, String aCardNumber)* as the constructor name. Click **Finish** to create the method feature.
4. Modify the code of the constructor:

```
public CheckingAccount(String anAccountId,String aCardNumber) {
    super(anAccountId, aCardNumber);
}
```

Repeat Steps 1 through 4 for `chap12.SavingsAccount`. The constructor you should write is:

```
public SavingsAccount(String anAccountId, String aCardNumber) {
    super(anAccountId, aCardNumber);
}
```

Beanizing the BankAccount Class

When you want to make an existing Java class behave as a bean, you can import it into the VisualAge for Java environment and choose among three approaches:

1. **Write an adapter class:** In this approach, you create a light-weight class that provides an interface between other beans and your existing class.
2. **Beanize your existing class manually:** In this approach, you manually add the event handling code to your class and then create a `BeanInfo` class that exposes the current property and method features of your class.
3. **Beanize your class automatically:** In this approach, you delete the class fields and re-create them as properties, using the `BeanInfo` page. In addition, you make public methods of your choices as method features by adding them to the `BeanInfo` class. In this case, you must ensure that the methods of your class refer to class fields by using their accessor methods.

In the sections that follow, you beanize the `BankAccount` class by using approach 3 for every field for which you want to re-create bound properties. In addition you modify the type of the `log` field to `IVector`, which is more suitable for displaying transactions in a list box, as you will see in “`AtmWelcomePanel`” on page 315.

Warning

Before starting modifying the Log field you must ensure that the `COM.ibm.ivj.javabeans.IVector` class is loaded in your workspace. Refer to Appendix A, “VisualAge for Java Installation,” on page 441 to load IVector.

Modifying the Log Field

Because you have to display the account transactions in the ATM view, it is a good idea to change the log type to `IVector`, which you can find in `COM.ibm.ivj.javabeans`. In effect, `IVector` has been especially designed to be used in association with the `IList` bean to display a list of elements in a list box. To modify the type of the log field:

1. Import the `COM.ibm.ivj.javabeans` package by using the import statement in the `BankAccount` class definition and change the type of log to `IVector` (in bold):

```
import java.util.*;
import java.io.*;
import COM.ibm.ivj.javabeans.*;

public abstract class BankAccount implements Serializable {
    private double balance;
    private String accountId;
    private static double initialAmount = 20.0;
    private IVector log;
}
```

2. Change both `BankAccount` constructors to reflect the new type of log (in bold):

```
public BankAccount(String anAccountId) {
    setAccountId(anAccountId);
    setBalance(0.0);
    setLog(new IVector());
}

public BankAccount(String anAccountId, String aCardNumber) {
    setAccountId(anAccountId);
    setLog(new IVector());
    setAtmCard(new AtmCard(aCardNumber,
        aCardNumber.substring(0, 2) + "123"));
}
```

Notice that the `BankAccount(String)` constructor has been slightly modified to call the setters methods of the `BankAccount` fields. This modification will take more sense once you have read “Migrating Fields to Bound Properties” on page 285.

3. Change the `getLog` method return type to `IVector`.
4. Change the `setLog` parameter type to `IVector`.
5. Modify the `addTransaction` method to reflect the use of an `IVector` instance variable (in bold):

```
public void addTransaction(Transaction aTransaction) {
    boolean found = false;

    for (int i = 0; (i < getLog().size()) && !found; i++) {
        Transaction trans = (Transaction)getLog().elementAt(i);
        if (aTransaction.before(trans)) {
            found = true;
            getLog().addOneElementAt(aTransaction, i);
        }
    }
    if (!found) {
        getLog().addOneElement(aTransaction);
    }
    return;
}
```

6. Save the changes in the `BankAccount` class by using the Save option from the `BankAccount` pop-up menu.

Migrating Fields to Bound Properties

If you open the Class browser with the `ch12.BankAccount` class and switch to the `BeanInfo` page, you should have the information shown in Figure 129.

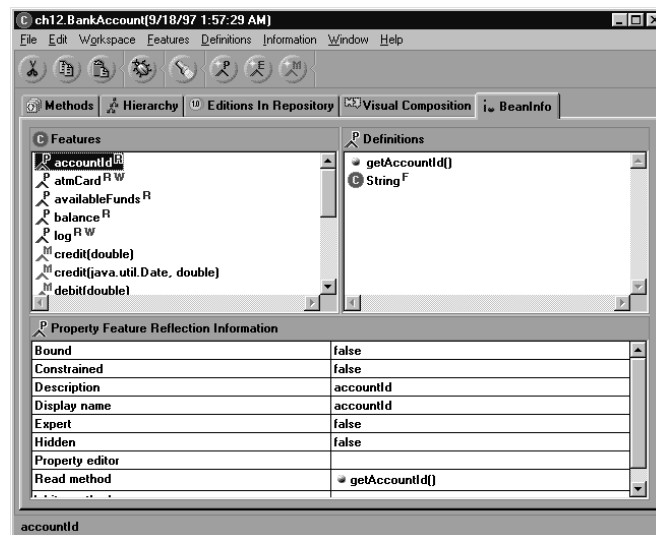


Figure 129. BeanInfo Page of the `BankAccount` Class

Because you used the syntax pattern suggested by the JavaBeans specification to code the setter and getter methods of `BankAccount`, and because a `BeanInfo` class is not associated with `BankAccount` yet, the `BeanInfo` page displays information that has been gathered by the reflection mechanism. (For information regarding the introspection and reflection process, refer to “Introspection and BeanInfo Class” on page 249). The following fields are listed:

- ❑ **accountId^R**: Because you define a public `getAccountId` method but no `setAccountId` method, the field appears as a read-only property feature.
- ❑ **atmCard^{RW}**: Because you define both `getAtmCard` and `setAtmCard` public methods, the field `atmCard` appears as a read-write property.
- ❑ **availableFunds^R**: Because you define a public `getAvailableFunds` method, `availableFunds` appears as a read property. Notice that you have defined no field for this property in the `BankAccount` class.
- ❑ **balance^R**: Because you define a public `getBalance` method but no `setBalance` method, the field appears as a read-only property feature.
- ❑ **log^{RW}**: Because you define both the `getLog` and `setLog` public methods, the field `log` appears as a read-write property.

In addition the `BeanInfo` page displays all public methods that are defined for the `BankAccount` class.

To support the event framework of the JavaBeans specification and facilitate the use of the `BankAccount` class in the Visual Composition Editor, you remove `accountId` and `balance` to re-create them as read-write bound properties. As you will see when you implement the ATM view, the other fields do not have to support the event framework.

To make the `accountId` a read-write-bound property, follow these steps:

1. Select the `accountId` field in the top left pane of the `BeanInfo` page and delete it by choosing the **delete** option from its pop-up menu. Make that both the `setAccountId` and `getAccountId` methods have been deleted by switching to the `Methods` page.
2. Delete the `accountId` field in the class definition from the `Methods` page. (A method should not be selected, to display the class definition in the source pane).
3. Once there is no trace of `accountId`, switch back to the `BeanInfo` page.

4. Select **Features**→**New Property Feature...** from the Bean-Info page menu bar.
5. Enter *accountId* for the Property name and *java.lang.String* for the Property type.
6. Make sure the **Writable**, **Readable**, and **bound** checkboxes are selected and click the **Finish>** button to create the property.

Once the property is created, you should have displayed in the BeanInfo page top-left pane the *accountId* with the RWB letters in superscript to confirm that the property is readable, writable, and bound. In the top-right pane, you should have displayed the get and set methods and the property type. Select the *setAccountId* setter method in the right pane. Its code is displayed in the source pane and should look like this:

```
/**
 * Sets the accountId property (java.lang.String) value.
 * @param accountId The new value for the property.
 * @see #getAccountId
 */
public void setAccountId(String accountId) {
    /* Get the old property value for fire property change event. */
    String oldValue = fieldAccountId;
    /* Set the accountId property (attribute) to the new value. */
    fieldAccountId = accountId;
    /* Fire (signal/notify) the accountId property change event. */
    firePropertyChange("accountId", oldValue, accountId);
    return;
}
```

Notice in the code that the implementation name of the field is *fieldAccountId* whereas the property name is *accountId*. This is the reason why you should always make sure that you refer to every field using its getter and setter methods when you write methods for Java classes that could later be beanized.

Notice also the notification of the property change with the call to the *firePropertyChange* method. By referring to the setter methods for each update of your fields, you ensure that events are propagated when you change those fields to bound properties.

Once you have created your first property from the BeanInfo page, VisualAge for Java creates its BeanInfo class for later use with the Visual Composition Editor.

To complete the BankAccount class modifications, repeat Steps 1 through 4 for the *atmCard* and *balance* fields.

Once all your modifications have been incorporated into the BankAccount class, the BeanInfo page should resemble Figure 130.

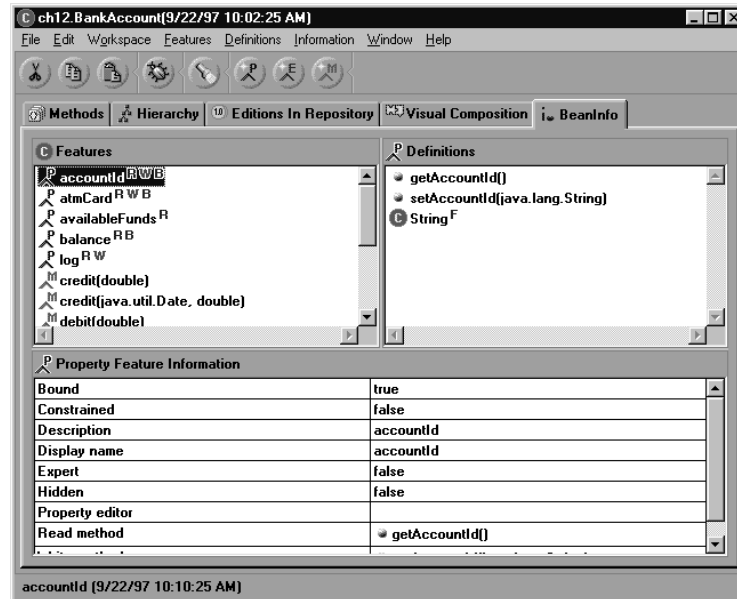


Figure 130. BeanInfo Page for the Updated BankAccount Class

Ordering BankAccount Objects

Using the ATM machine, the user will have access to several accounts. To order these accounts by their identifier, you complete the BankAccount class by adding a before() method:

1. Select the ch12.BankAccount class in the Workbench and open the Class Browser.
2. From the Methods pull-down menu, select **New Method...** and add the public method, *void boolean before(BankAccount anAccount)*.
3. Click **Finish>** to create the method.
4. Modify the before method as follows:

```
public boolean before(BankAccount anAccount) {
    int result = getAccountId().
        compareTo(anAccount.getAccountId());
    return (result <= 0) ? true : false;
}
```

5. Save the BankAccount class.

Modifying the toString Method

A dedicated panel will be designed to display the transactions of a selected bank account. For this reason, you do not have to return the transactions of the account in its toString method. Rather, you return only its account identifier:

```
public boolean toString() {  
    return getAccountId().  
}
```

Creating a Bank Bean

Now that you have created the AtmCard class and modified the BankAccount class, you have to create a Bank class to hold several accounts. For this purpose you create a Bank class that holds a field of type IVector for the list of all bank accounts managed by the bank. The Bank class also holds a name that identifies it. Besides the accessor methods associated with its fields, the Bank class has an addAccount method to add new bank accounts in the IVector field, an initialize method to create several accounts from memory, and a toString method to display it.

Figure 131 represents the Bank class with its fields and methods.

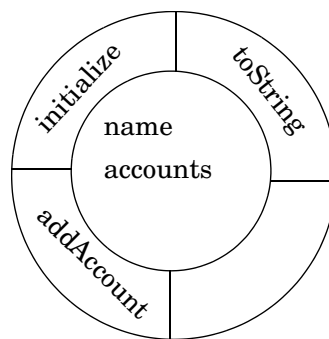


Figure 131. Bank Class

Because you will connect a Bank object to the ATM view, you create the Bank class as a bean by using the BeanInfo page as described in “The BeanInfo Page” on page 252.

As with the creation of the AtmCard class, you create the bean in three steps: you create the Java class, you add property features, and you add method features.

Creating the Bank Class

To create the Bank class using the Create Class SmartGuide, follow these steps:

1. Start the Create Class SmartGuide to create your Bank class.
2. Fill in the Class Name entry field with *Bank* and check the **Browse the class when finished** checkbox. Then click the **Next>** button.
3. Click the **Add Package...** button and select the *java.io* package. Then click the **OK** button. Repeat the operation for the *Com.ibm.ivj.javabeans* package.
4. Check the **toString()** checkbox to generate a method stub and click the **Next>** page to specify an interface to implement.
5. Click the **Add...** button to specify an interface and choose *java.io.Serializable*.
6. Click the **Finish>** button to create the Bank class and open the Class Browser window.

Adding Property Features

Now that the class is created, you code it as a bean by adding property and method features. To add property features, follow these steps:

1. Switch to the BeanInfo page to specify the new features and ensure that a BeanInfo class will be generated for your Bank bean.
2. Create a new Property Feature using **Features→New Property Feature....** Name it *accounts*. Click **Browse** for the property type and select *IVector*. Make sure that the **Readable**, **Writable** and **bound** checkboxes are checked (Figure 132). Click **Finish** to create the property.
3. Repeat Step 8 to create a *name* property of type *String*.

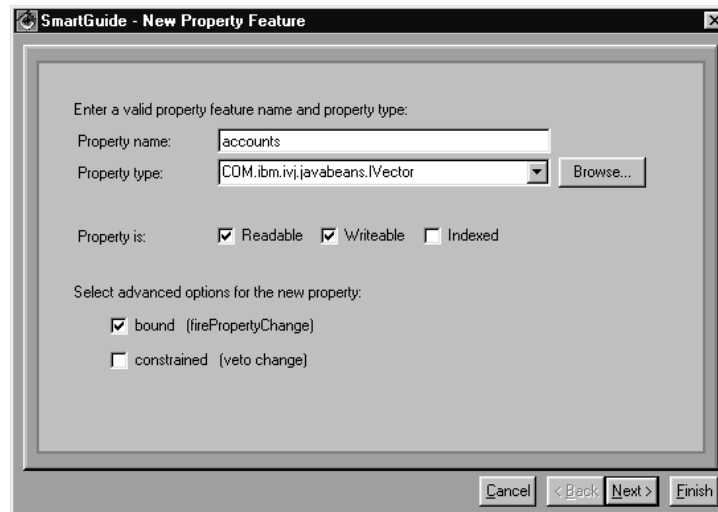


Figure 132. Accounts Property Creation

Adding Method Features

Notice that because you created name and accounts as property features, VisualAge for Java generates the accessor methods that comply with the JavaBeans specification. To complete the creation for your bean, you have to add an *addAccount* method, which adds an account to the vector field, and an *initialize* method feature, which initializes the bank object by adding several accounts:

1. Create a new Method Feature using **Features**→**New Method Feature....** Enter *addAccount* as the method name, *void* as the return type, and *1* for the parameter count. Click **Finish** to create the method feature.
2. Modify the code of the method feature:

```
public void addAccount(ch12.BankAccount anAccount) {
    boolean found = false;
    COM.ibm.ivj.java.beans.IVector list = getAccounts();
    for (int i = 0; i < list.size() && !found; i++) {
        BankAccount acct = (BankAccount) list.elementAt(i);
        if (anAccount.before(acct)) {
            found = true;
            list.addOneElementAt(anAccount, i);
        }
    }
    if (!found) {
        list.addOneElement(anAccount);
    }
    return;
}
```

3. Create a new Method Feature using **Features**→**New Method Feature...**. Enter *initialize* as the method name, *void* as the return type, and *0* for the parameter count. Click **Finish** to create the method feature.
4. Modify the code of the method feature as follows:

```
public void initialize() {  
    for (int i = 3; i > 0; i--) {  
        BankAccount checkAccnt =  
            new CheckingAccount("0" + i + "-000-" + (i + 1) + "0",  
                                "0" + i + "0" + 2*i);  
        BankAccount saveAccnt =  
            new SavingsAccount("1" + i + "-000-" + (i + 1) + "0",  
                               "1" + i + "0" + 2*i);  
        addAccount(checkAccnt);  
        addAccount(saveAccnt);  
        return;  
    }  
}
```

In this initialize method feature, you create six accounts. The first three are checking accounts and are identified by account numbers starting with 0. The last three are savings accounts and are identified by account numbers starting with 1.

Creating the ATM View

Now that your model is updated, you can create the ATM view to interact with it. The ATM view is a representation of a real-world ATM machine.

When the user approaches the ATM, a first panel is presented as shown in Figure 133. This panel displays a list box that contains the accounts (checking or savings) of the bank, a start button to start new transactions on a selected account, and two fields at the top that display the identifier and the associated ATM card number of the selected account.

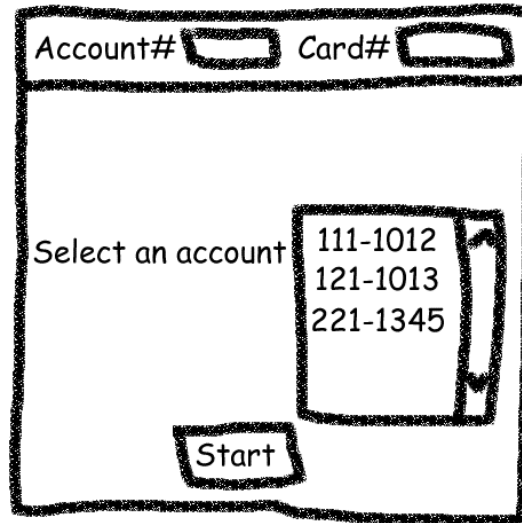


Figure 133. ATM Welcome Panel Sketch

Once the user selects an account and validates by clicking the **Start** button, a second panel is displayed and prompts the user to enter the PIN for the associated ATM card (Figure 134). The user can type in the PIN, using a dedicated keyboard with 12 keys. The C key clears the PIN entered.

If the user enters a wrong PIN and clicks the **OK** button, nothing happens, and the entry field that contains the entered PIN is cleared.

If the user enters the correct PIN and clicks the **OK** button, the ATM displays the next panel.

If the user clicks the **Cancel** button, the first panel is displayed again to let the user choose another account.

Account# 111-1012 Card# 111

Please enter your PIN number

1	2	3
4	5	6
7	8	9
0	.	C

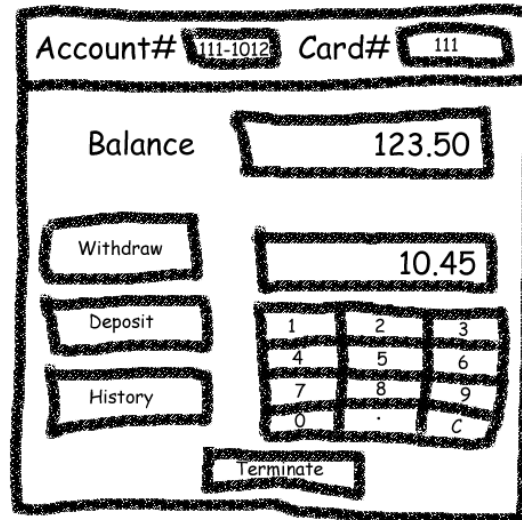
OK Cancel

Figure 134. ATM PIN Panel Sketch

From the third panel, the user can create transactions on his or her account by making a withdrawal or a deposit. The user enters the amount of money for the transaction by using a numeric keyboard similar to the keyboard to enter the PIN. Once the transaction is completed, the current balance of the account is updated and shown on the top-most entry field (Figure 135).

The current system date and time are displayed on the left side of this panel. This information is recorded with the transaction and can be displayed in the next panel.

If the user clicks the **Terminate** button, the first panel is displayed to select another account.

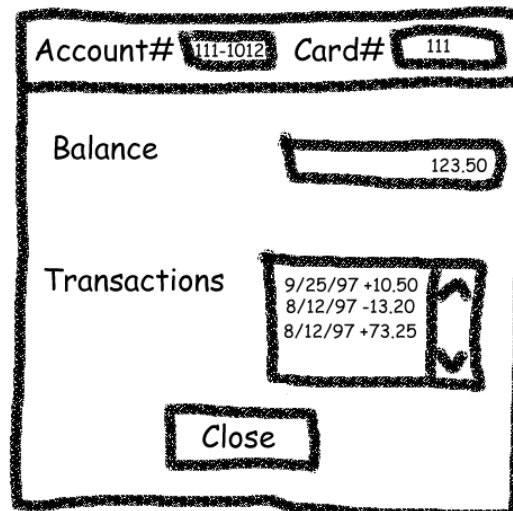


A hand-drawn sketch of an ATM balance panel. At the top, it shows 'Account#' with the value '111-1012' and 'Card#' with the value '111'. Below this, the word 'Balance' is followed by a rectangular display showing '123.50'. To the left of the panel are three buttons: 'Withdraw', 'Deposit', and 'History'. To the right of the 'Balance' display is another rectangular display showing '10.45'. Below the 'Deposit' button is a numeric keypad with buttons for digits 1-9, 0, a decimal point, and a 'C' (clear) button. At the bottom center is a 'Terminate' button.

Figure 135. ATM Balance Panel Sketch

If the user clicks the **History** button, the last panel is displayed and shows the transaction list for the selected account alongside the current balance (Figure 136).

For each transaction the date and amount are displayed. A **Close** button enables the user to switch back to the previous panel to create new transactions.



A hand-drawn sketch of an ATM panel showing both balance and transactions. At the top, it shows 'Account#' with the value '111-1012' and 'Card#' with the value '111'. Below this, the word 'Balance' is followed by a rectangular display showing '123.50'. Below the balance display is the word 'Transactions' followed by a rectangular display containing a list of transactions: '9/25/97 +10.50', '8/12/97 -13.20', and '8/12/97 +73.25'. To the right of the transaction list is a vertical scroll bar. At the bottom center is a 'Close' button.

Figure 136. ATM Balance and Transactions Panel Sketch

In the sections that follow you create the ATM panels, using the Visual Composition Editor. To reuse the composite beans within your application, you identify the bean you can reuse from one panel to another. Once those beans are identified, you list all the composite beans you need to create your interface, and you arrange them in a tree representation that ordinales the bean creation. Beans at the bottom of the tree have to be built first. In this way you start to build the simplest beans and you work up to the more complex beans.

Reusing Composite Beans

Looking at Figures 133 through 136, you see that the top part, which contains the account identifier and the ATM card number, occurs on each panel. This is a good candidate for a reusable bean. We decide to call this bean *AccountPanel*.

If you compare Figure 134 with Figure 135, you notice that the keyboard and its entry field used to enter the card PIN can also be used to enter the amount of the transactions. Again, this bean is a good candidate for reuse. We decide to call it *AtmKeypadPanel*. This *AtmKeypadPanel* is made up of the *AtmKeypad*, which represents the keypad itself (Figure 137).

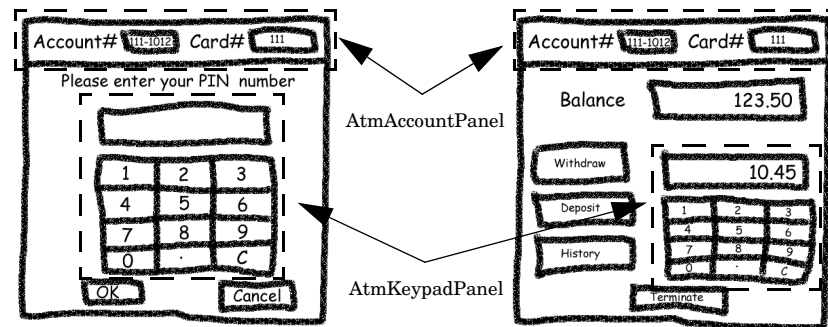


Figure 137. Reusing Visual Beans

Organizing the Building of Your Beans

Now that you have identified *AtmAccountPanel* and *AtmKeypadPanel* as reusable beans, you can flesh out the dependency tree of the beans that constitute the ATM user interface.

To build the first panel (Figure 133), you need to assemble the *AtmAccountPanel* with an *AtmWelcomePanel*, which displays the account list.

The second panel of the ATM machine (Figure 134) is made up of the *AtmAccountPanel* and the *AtmPinPanel* which displays the keypad. This panel is made itself from the *AtmKeypadPanel*, which is made from the *AtmKeypad*.

The third panel (Figure 135) is built up from the *AtmPanel* and the *AtmAccountPanel*.

The last panel (Figure 136) is built up from the *AtmHistoryPanel* and the *AtmAccountPanel*.

Figure 138 shows the dependency tree for the different composite beans that you are going to build.

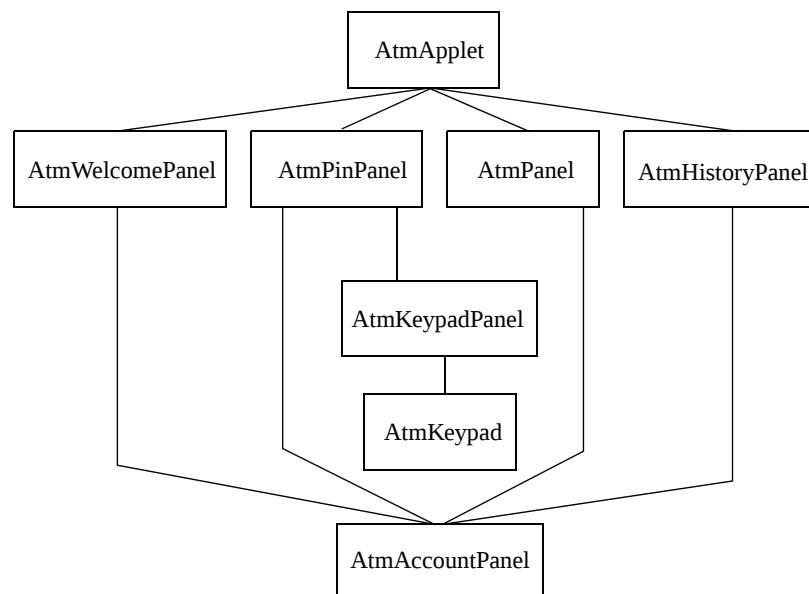


Figure 138. Composite Bean Dependency Tree

In the next sections you build each visual bean, using the Visual Composition Editor. Throughout your building, you will discover how you can use each type of layout manager with the Visual Composition Editor. In addition, you will learn about variable beans and reinforce your knowledge of the different connections you can use to build your application.

Using Layout Managers in Visual Composition

AWT container components support the use of layout managers, classes that implement the *java.awt.LayoutManager* interface. Layout managers control the placement, initial sizing, and run-time resizing behavior of components within the container.

You assign a layout manager, which governs placement, to the container. In most layouts, you can then attach layout properties to the components in the layout. These properties govern the initial sizing and resizing behavior for the components.

If you want to use a layout that is not shipped with AWT, you can write your own layout manager implementation. However, custom implementations are not supported for visual composition in the current release.

The use of visual composition makes trying out different layout alternatives relatively easy. If you prefer to lay out beans individually, you can stick with null layout (that is, no layout) and use the Visual Composition Editor's alignment tools instead as you did with the Calculator view in Chapter 11.

Supported Layouts

VisualAge supports the use of different layouts in container beans. For several of these layouts, VisualAge sets layout properties by default. The layouts are associated with the layout managers listed below:

- ❑ **BorderLayout** arranges a component along each edge of the container. A fifth component can reside in the center of the container. If your base bean uses a border layout, another bean placed as the center component will expand to fill all empty space. The beans list allows you to easily perform tasks on the covered components. You will use this layout manager to build *AtmKeypadPanel* and *AtmPanel*.
- ❑ **CardLayout** arranges components in a linear depth sequence similar to a notebook or tabbed dialog. Each component is called a *card*. You can use **Switch To** on the pop-up menu to move through the deck or perform tasks on the covered cards in the beans list. You will use this layout manager to build *AtmApplet*.
- ❑ **FlowLayout** arranges components in horizontal lines. The alignment property in the layout manager enables you to specify where you want the flow to begin. You will use this layout manager to build *AtmPanel*.

- ❑ **GridLayout** arranges components in a table, with all cells having the same size. You will use this layout manager to build `AtmPanel`.
- ❑ **GridBagLayout** enables you to arrange components in a highly complex grid. As you add or move beans, the free space shuffles so that beans are centered on the interface, while your arrangement is retained. Grid cells are not necessarily identical in size, and components can span multiple cells. You can customize grid sizing behavior down to each individual component. You will use this layout manager to build `AtmAccountPanel`, `AtmWelcomePanel`, `AtmPanel`, and `AtmHistoryPanel`.
- ❑ **Null layout** means that no layout manager is assigned. Without a layout manager, resizing the container at run time does not affect the size and position of the components. You can customize components within the null layout by means of dragging the beans, using Tool Bar options, or through the constraints option in Properties. This is the option you used in Chapter 11 to build the calculator view. Alignment tools are disabled for all but null layout.

Setting Layout Properties during Visual Composition

For most layouts, you can specify spacing between adjacent components. In most cases, these property values match the API. For details, see the Java API documentation.

For border and card layouts, additional settings are not necessary. For flow layout, you can specify for alignment to start at center, left, or right. For grid layout, you can specify the initial number of rows and columns. For grid bag layout, you specify most properties using the same units as shown in the API. The exception is the insets properties, which determines the gutter between adjacent cells. The initial value of insets is `T0 B0 L0 R0`. Read this as "top, 0; bottom, 0; left, 0; right, 0." All units are pixels.

Consider waiting to set layout properties until you have settled on a layout manager. Many values are lost when you switch layouts or move the component to another container on the free-form surface.

Dropping Beans into the Layout

Once you have assigned a layout, the Visual Composition Editor provides visual cues to help you place beans in the correct position. These cues appear when you place the loaded mouse pointer in position and then press and hold mouse button 1:

- ❑ For flow and grid layouts, a bold vertical bar appears. If you release the mouse, VisualAge places the bean to the right of the vertical bar.
- ❑ For border and null layouts, an outline of the bean appears. When you release the mouse, VisualAge places the bean at the outline.
- ❑ For grid bag layout, a bold grid appears that is based on the beans dropped so far. VisualAge attempts to place the bean as indicated by the pointer. If the pointed-to cell is empty, all borders of the cell are highlighted. If the pointer rests on a row or column boundary, a new row or column is inserted. New rows appear below the pointer; new columns appear to the right.
- ❑ For card layout, there are no visual cues. VisualAge adds beans to the top of the card deck, making the first bean you dropped the bottom card. You can use **Switch To** on the pop-up menu to move through the deck, or perform tasks on the covered cards in the beans list. To get access to the layout interface directly, drop a Variable bean on the free-form surface to the right of the container. Change the type of the Variable bean to that of the class implementing the layout manager interface (for example, `CardLayout`). Connect the layout property of the container bean to the *this* property of the Variable bean. Then connect to features of the Variable bean.

If you use a layout that allows for a bean to completely cover another bean, the beans list enables you to easily perform tasks on the covered components. To modify bean placement on the Visual Composition Editor from within the beans list, open the Properties for the bean and modify the Constraints.

In the sections that follow you build each visual bean of the ATM applet and experiment with the various layout managers.

AtmAccountPanel

`AtmAccountPanel` is a simple panel that displays an account identifier and its ATM card number. The panel is associated with a *GridBagLayout* manager to ensure that both text fields are expanded when the panel is resized. Although this composite bean

is simple, it will introduce you to some fundamentals of visual programming: setting tabbing order, using variables, tearing bean features off, and promoting bean features.

To create `AtmAccountPanel` follow these steps:

1. Open the Workbench window.
2. Select the *ch12* package inside the *Learn Java* project.
3. Create an *AtmAccountPanel* class that inherits from *java.awt.Panel*. Make sure the **Design the class visually** radio button is selected.
4. Click **Finish>** to create the class and load it in the Visual Composition Editor.
5. Drop two labels and text fields as shown in Figure 140.
6. Change the text attribute of each label to match the text of Figure 140.
7. Rename the first label to *AccountLabel* and the second to *CardLabel*.
8. Rename the first text field to *AccountTextField* and the second to *CardTextField*.
9. Change the editable property of the box text field to *false* to prevent the user from changing its contents.
10. Double-click the panel to open its property sheet.
11. Change the layout property to *GridBagLayout* to associate a new layout manager with *AccountPanel*. The labels and text fields are repositioned. During the next manipulations, keep the property sheet open.
12. Select the left-most label. The property sheet changes to display the label properties.
13. Open the settings of the constraints property by clicking the button in the right field. The Prompter window is displayed (Figure 139), where you can set the constraints of the label (see “GridBagLayout” on page 221 for constraint settings information).

14. Set the label constraints as follows:

- anchor: *WEST*
- fill: *NONE*
- gridHeight: *1*
- gridWidth: *1*
- gridX: *0*
- gridY: *0*
- insets: *T5 L5 R5 B5*
- ipadX: *0*
- ipadY: *0*
- weightX: *0.0*
- weightY: *0.0*

15. Set the constraints for the other controls as shown in Table 9. Then close the property sheet.

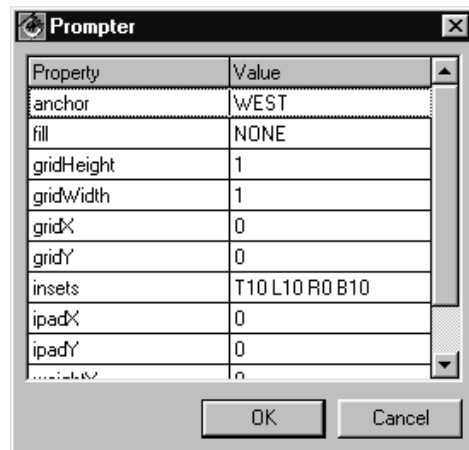


Figure 139. Prompter Window

Table 9. AtmAccountPanel GridBagLayout Constraints

Property	Account-Label	Account-TextField	CardLabel	CardText-Field
anchor	WEST	WEST	EAST	EAST
fill	NONE	HORIZON-TAL	NONE	HORIZON-TAL
gridHeight	1	1	1	1
gridWidth	1	1	1	1
gridX	0	1	2	3

Table 9. AtmAccountPanel GridBagLayout Constraints

Property	Account-Label	Account-TextField	CardLabel	CardText-Field
gridY	0	0	0	0
insets	T10 L10 R0 B10	T10 L0 R10 B10	T10 L10 R0 B10	T10 L0 R10 B10
ipadX	0	0	0	0
ipadY	0	0	0	0
weightX	0.0	1.0	0.0	1.0
weightY	0.0	0.0	0.0	0.0

Notice that only the two text fields are expandable. You can resize the panel on the free-form surface to check the expected behavior.

**Figure 140.** AtmAccountPanel

Setting the Tabbing Order

The tabbing order specifies how the input focus moves from bean to bean as the user presses the Tab key.

To display the tabbing order, open the pop-up menu for the composite bean. Select **Set tabbing** and then **Show tab tags**. Numbered tab tags appear as shown in Figure 141.

**Figure 141.** Tabbing Order in AtmAccountPanel

To change the tabbing order:

1. Open the pop-up menu for the composite bean.
2. Select **Set tabbing** and then **Show tab tags**. Numbered tab tags appear.
3. Place the mouse pointer over the tab tag you want to change.

4. Press and hold mouse button 1.
5. Drag the tab tag to its new position.
6. Release mouse button 1.
7. Continue until the numbered tags reflect the tabbing order you desire.

You can also change the tabbing order by moving the beans in the Beans List as shown in Figure 142.

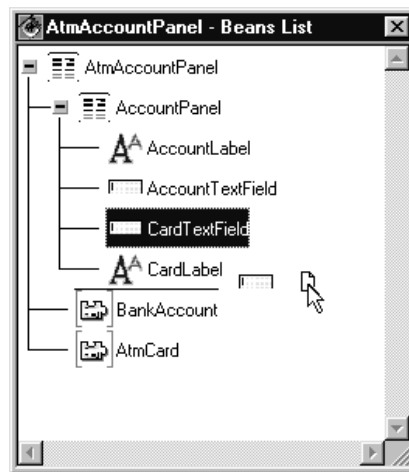


Figure 142. Changing Tabbing Order in Beans List

Once you have set the tabbing order as shown in Figure 141, you can save and test your AccountPanel by clicking the left-most icon from the toolbar.

Using Variable Beans

Variable beans play an important role in application or applet programming. You use variable beans primarily in two situations:

- ☐ To act as a place holder inside a composite bean for beans that cannot be found in the composite bean. Using variable beans enables you to pass data or functions between beans.
- ☐ To represent bean instances created with factory beans.

You use variable beans to hold the BankAccount object the user has selected from the AtmWelcomePanel list box. As you will see in “Putting Your Applet Together” on page 329, this account is passed to AtmAccountPanel through a variable of type BankAccount.

You can add a variable bean on the free-form surface in three different ways:

- ❑ Select a variable bean from the Models category on the bean palette. Once the variable bean is on the free-form surface, you must change its type to the type of the bean it represents.
- ❑ From the Visual Composition Editor menu, use **Options** → **Add Bean...** to add a bean. Instead of selecting the **Class** radio button in the Bean Type group box, you select the **Variable** radio button. In this case the type of the variable bean is automatically set according to the bean class you have entered.
- ❑ Use the tear-off feature to expose a feature of a bean as explained in “Tearing Off Bean Features” on page 306. In this case the type of the variable bean is automatically set according to the class of the bean exposed.

Add a variable of type `BankAccount`, using either of the first two ways. To display the account identifier in the `AccountTextField`, make a property-to-property connection between the `accountId` property of the variable and the text property of the text field (Figure 143). Notice that because `accountId` is a bound property of `BankAccount`, it is shown in the `BankAccount`’s pop-up menu.

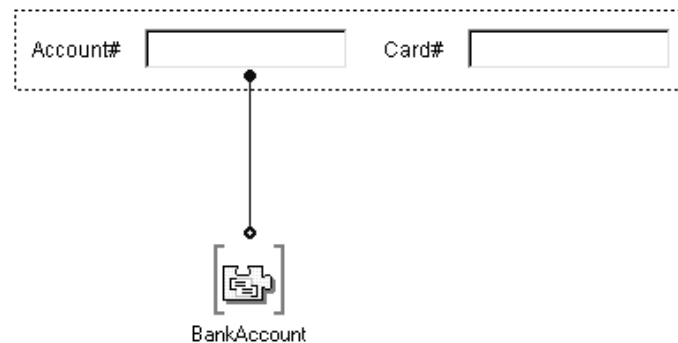


Figure 143. Connecting a `BankAccount` Variable to `AccountTextField`

Firing Events from AWT Beans

If you double-click the property-to-property connection to open its property sheet (Figure 144), you will notice that the Source event drop-down list box is set to `accountId`, whereas the Target event drop-down list box is set to `<none>`. Therefore if the `accountId` property is changed, the associated event fires the connection. But, if the text property of the text field is changed, an event is not sent, and the connection is not fired. The property-to-property connection is in this case unidirectional.

By compatibility with Sun AWT classes, primitive beans, which are the mappings of the AWT classes, do not fire events when their properties change. However, IBM has provided these beans with events that can be fired when their properties change. If you want to fire a property event for the AWT bean and trigger an associated connection, select the event from the Source event or Target event drop-down list boxes of the connection property sheet. For example, if you want to fire the property-to-property connection from AccountTextField to BankAccount when the text property of AccountTextField changes, select the *text* event in the Target event drop-down list box.

Notice that, because you set AccountField readable only, the user cannot update this field. Therefore only fire the property-to-property connection from the BankAccount variable to AccountTextField.

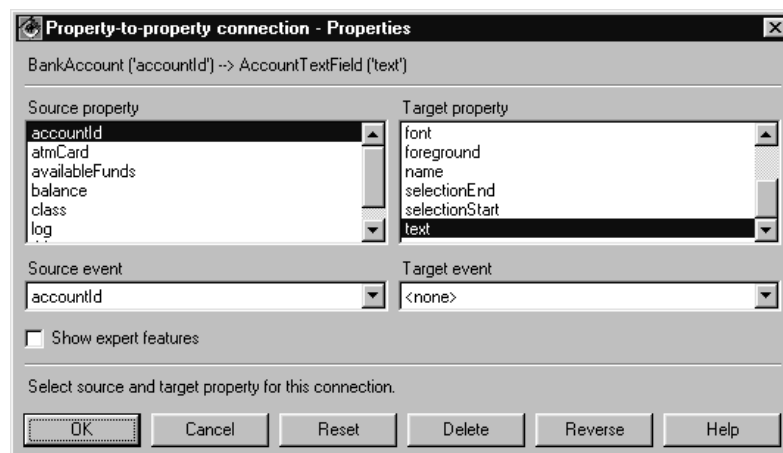


Figure 144. Property-to-Property Connection Property Sheet

Tearing Off Bean Features

You tear off a property to gain access to the encapsulated features of a bean. This can be necessary when a property is itself a bean and you want to connect to one of its features. The torn-off property is not actually a separate bean but a variable that represents the property itself or points to it. For example, to populate CardTextField, you have to access the ATM card number associated with the account and make a property-to-property connection between the card number and the text property of CardTextField.

By tearing off the AtmCard property feature of the BankAccount variable, you can access the account card number :

1. Select the BankAccount variable.
2. Select **Tear-Off Property...** from its pop-up menu. The Tear-Off Property Dialog is displayed (Figure 145).
3. Select *atmCard(AtmCard)* to tear off the property atmCard of type AtmCard. A variable, atmCard1, is added on the free-form surface and a property-to-property connection is created from the atmCard property of the BankAccount variable to the *this* property of the atmCard1 variable (see 1 in Figure 146). If you want, you can change the AtmCard variable to AtmCard instead of atmCard1.



Figure 145. Tear-Off Property Dialog

Once the atmCard property feature is available, you can access the card number and make a property-to-property connection from the cardNumber property of the AtmCard variable and the text attribute of CardTextField as shown in Figure 146:

1. From the AtmCard variable pop-up menu, select **Connect→All Features...**
2. Select *cardNumber^R* from the Property list box.
3. Complete the connection by connecting to the *text* attribute of the CardTextField (see 2 in Figure 146).

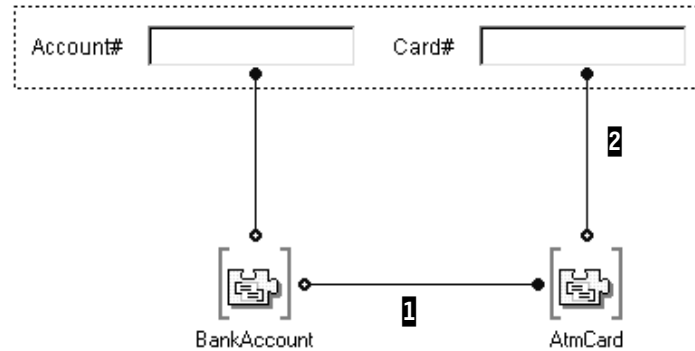


Figure 146. Populating CardNumber in CardTextField

Promoting Bean Features to a Composite Bean Interface

When you create a composite bean, you might want some features of beans embedded within it to appear in its interface. When you first create a composite bean, its interface reflects its inheritance, not the features of the beans embedded within it. To expose features of the embedded beans, you must promote them to the composite's interface. For example, if you want to reuse `AtmAccountPanel`, you need to promote the `BankAccount` variable feature. In this way, when you add the `AtmAccountPanel` composite bean to another composite, you can connect to the `BankAccount` variable and transmit an account object, which will be used to populate `AccountTextField` and `CardTextField`.

Alternatively, you can promote the *this* property of an embedded bean, which represents the bean itself, to add the entire bean as a property of the composite bean. Then, when you use the composite bean in another composite bean, you can tear off the property as a `Variable` bean and connect to features of the embedded bean represented by the `Variable`.

To promote the `BankAccount` variable, you promote its *this* feature:

1. From the pop-up menu of the `BankAccount` variable, select **Promote Bean Feature** to open the Promote Features window.
2. Select the *this* feature from the Property list. You can use the default composite bean feature name that appears in the Promote feature name field, or you can change the name. The default feature name is a combination of the name of the bean you are promoting from and the name of the feature you are promoting. This identifies the bean that implements the fea-

ture, which is helpful if the composite bean contains more than one bean with the same feature. Then, when you connect to the feature, you can tell to which embedded bean it belongs. For example, if you select the *this* feature of the BankAccount variable named BankAccount, the default feature name is *bankAccountThis*.

3. Click **Promote**. The feature name is added to the Previously promoted list (Figure 147).
4. Click **OK** to close the Promote Features window.
5. Save the composite bean to incorporate the features you just promoted. If you run the test tool, the bean is automatically saved.

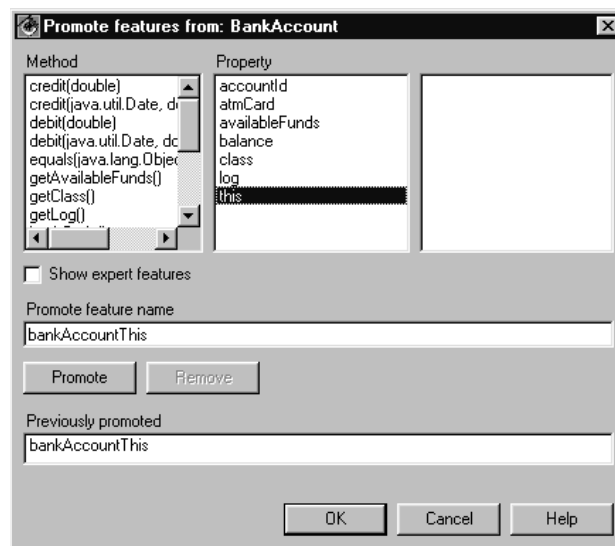


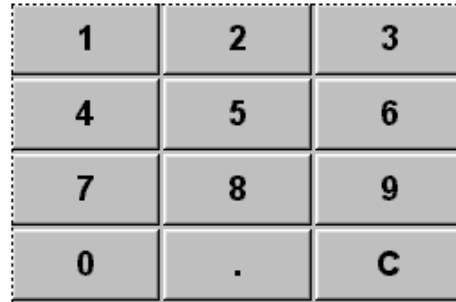
Figure 147. Promoting a Feature



Congratulations! You have just built your first reusable ATM panel and you have learned most of the tricks you will use to build the other panels. Now let's build the rest of the visual beans.

AtmKeypad

AtmKeypad provides the user with a way of typing a number. This number is recorded in a property of the visual bean itself. AtmKeypad is built from a panel associated with a *GridLayout* manager. This layout manager serves to display the 12 buttons that constitute the keypad (Figure 148).



1	2	3
4	5	6
7	8	9
0	.	C

Figure 148. AtmKeypad

To create `AtmKeypad`, follow these steps:

6. Create an `AtmKeypad` class in the `ch12` package that inherits from `java.awt.Panel`. Make sure the **Design the class visually** radio button is selected.
7. Click **Finish>** to create the class and load it in the Visual Composition Editor.
8. In the Class Browser, select the BeanInfo tab, to add a new property feature to your class.
9. Select **Features→New Property Feature** in the menu bar.
10. Type **value** as the property name and **String** as its type. Make sure the **Readable**, **Writable**, and **bound** checkboxes are selected.
11. Click **Finish>** to create the property.
12. Switch to the Visual Composition Editor.
13. Select the empty panel on the free-form surface and open its property sheet.
14. Change the layout property to `GridLayout` (3 columns, 4 rows, `vgap` and `hgap` of 2).
15. Change the font point size to *18* and the style to *Bold*. This way, each button you drop on the panel will inherit this setting for its label.
16. Select the button bean in the Beans palette, click the **Sticky** checkbox, drop 12 buttons on the panel, and change their labels according to Figure 148.
17. Save the part to generate the code.

Right now, you can test your bean and verify that you can press each button. However, the key strokes are not recorded as a value in the value property of the bean. To save the keys that are pressed, you need to capture the `ActionEvent` sent by the button

and adjust the value property accordingly. Because you decided not to expose the way the value is adjusted, you develop a script method that will be called each time a key is pushed:

1. In the Visual Composition Editor, connect the *ActionEvent* (listed as *actionPerformed(java.awt.event.ActionEvent)* in the button pop-up menu) of the button 1 to the free-form surface and select **Event to script...**
2. Click the **New method...** button to create a new script method.
3. Enter *void recordValue(java.awt.event.ActionEvent)* as the method name and select **public** as the access modifier. Then click **Finish>** to create the method.
4. Click **OK** to create the script connection (refer to “Script Connections” on page 263).
5. Double-click the script connection and check the **Pass event data** checkbox to pass the *ActionEvent* sent by the button to the script method (Figure 149). By passing the event, you will be able to know which button has been pressed and adjust the value property accordingly.
6. Create 11 more script connections for the other buttons and connect every connection to the *recordValue* method (Figure 150).
7. Switch to the **Methods** page and select the *recordValue* method. Then modify its implementation:

```
public void recordValue(java.awt.event.ActionEvent e) {
    String label = ((java.awt.Button)e.getSource()).getLabel();
    if ((label.compareTo("0") >= 0) &&
        (label.compareTo("9") <= 0)) {
        setValue(getValue()+label);
    }
    else if (label.compareTo(".") == 0 &&
        getValue().indexOf(".") == -1) {
        setValue(getValue()+label);
    }
    else if (label.compareTo("C") == 0) {
        setValue("");
    }
    System.out.println("Value = " + getValue());
    return;
}
```

After saving the changes, you can switch back to the Visual Composition Editor to test your keypad.

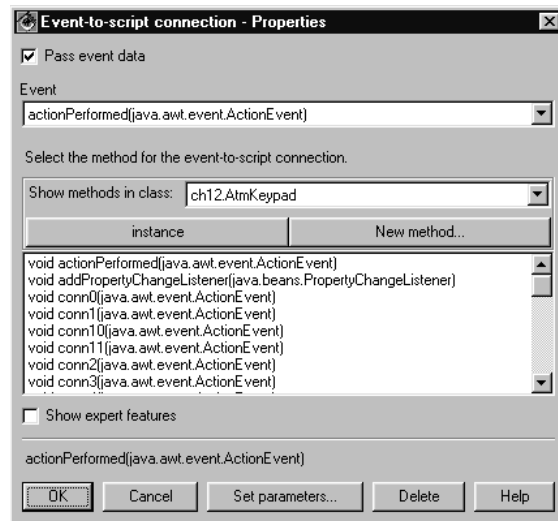


Figure 149. Event-to-Script Connection - Properties Window

Notice that a `println` call is inserted in the `recordValue` method to print out the property value on the Console window. When you have checked that the keypad runs as expected, you can remove this line.

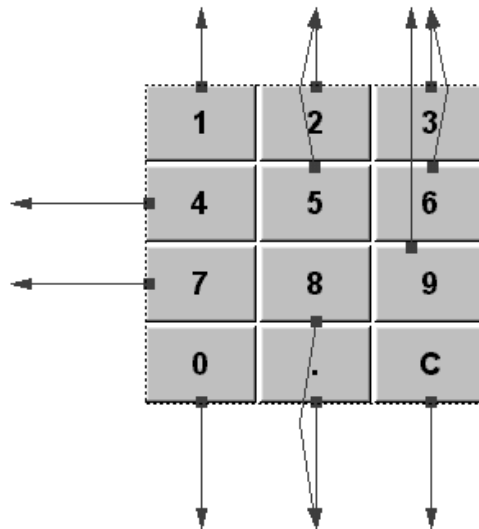


Figure 150. AtmKeypad with Script Connections

Now that the `AtmKeypad` bean is completed, you can start developing the `AtmKeypadPanel`.

AtmKeypadPanel

`AtmKeypadPanel` is built from `AtmKeypad` and adds a read-only text field to display the `AtmKeypad` value. A clear method allows you to clear the text field by setting the `AtmKeypad` value to a null string. This time you use a *BorderLayout* manager to place your beans. You may wonder why we did not provide `AtmKeypad` with a text field at first. We could have, but we decided to create an `AtmKeypad` that we could reuse as a simple keypad.

To create `AtmKeypadPanel`, follow these steps:

1. In the `ch12` package create an *AtmKeypadPanel* class that inherits from *java.awt.Panel*. Make sure the **Design the class visually** radio button is selected.
2. Click **Finish>** to create the class and load it in the Visual Composition Editor.
3. Double-click the panel to open its property sheet.
4. Change the layout manager to *BorderLayout*.
5. Drop a text field to the north region of the panel. Notice the visual feedback that is displayed when you drag your text field over the panel (Figure 151).

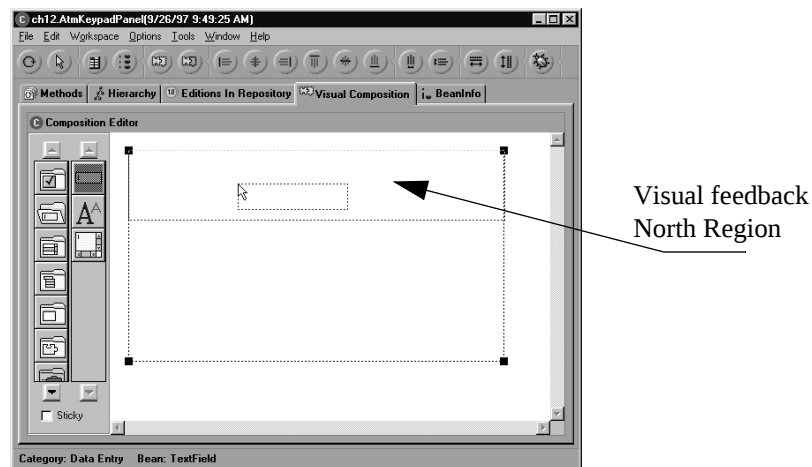


Figure 151. Visual Feedback from a BorderLayout

6. Change the text field name to *ValueTextField*.
7. Change the text field font size to *18* and the font style to *Bold*.
8. Change the text field editable property to *false* to prevent users from typing in it.

9. Connect the *value* property of the *AtmKeypad* to the *text* attribute of the text field, to display the value in the text field (Figure 152).
10. Save the bean by pressing Ctrl+F2 or by selecting **File**→**Save Bean**. By saving the bean you generate methods such as *setAtmKeypadValue* that you call in Step 12.
11. Switch to the Methods page to create the clear method.
12. Enter *void clear()* as the method name and select public as the access modifier. Then click **Finish**> to create the method.
13. Modify the clear method implementation:

```
public void clear() {
    setAtmKeypadValue("");
    return;
}
```

14. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

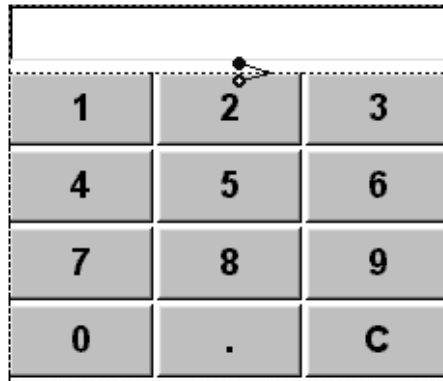


Figure 152. *AtmKeypadPanel*

Notice that, because *value* is a public property feature of *AtmKeypad*, it does not have to be promoted to be accessible from *AtmKeypadPanel*.

Reusing AtmKeypadPanel in the Applet

You reuse *AtmKeypadPanel* in the applet to enter the PIN in *AtmPinPanel* and the transaction amounts in *AtmPanel*. When using *AtmKeypadPanel* to enter the PIN, you want make sure that the number remains confidential. For this reason, you should not echo the characters in *ValueTextField* but echo a dummy character, such as a star, instead.

To set the echoChar property feature of AtmKeypadPanel depending on the composite part where it is embedded, you need to promote this feature. Then when you embed AtmKeypadPanel in AtmPinPanel, you will set echoChar to “*”.

To promote the echoChar feature, follow these steps:

1. Select ValueTextField and from its pop-up menu choose **Promote Bean Feature...**
2. Select echoChar in the Property list of the Promote features from:ValueTextField window (Figure 153) and accept the default name: ValueTextFieldEchoChar.
3. Click **OK** to validate your change.
4. Save AtmKeypadPanel.

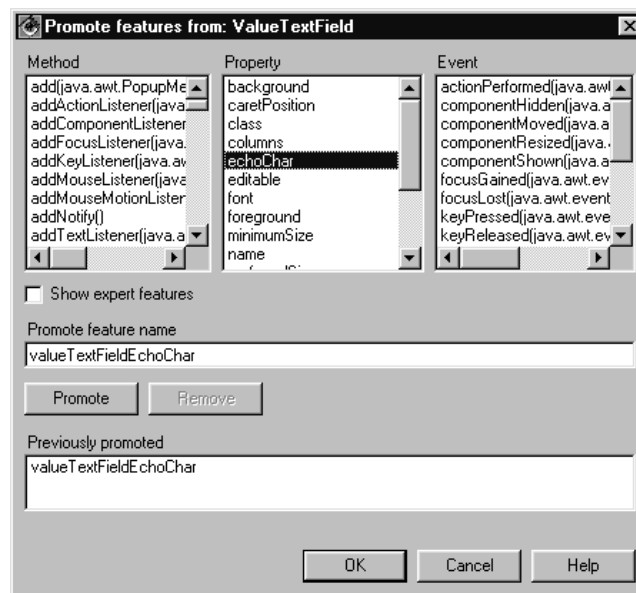


Figure 153. Promoting the EchoChar Property Feature

AtmWelcomePanel

AtmWelcomePanel is the first panel displayed by the ATM. It allows the user to select an account and access the AtmPinPanel. AtmWelcomePanel is built from an IList bean, which lists the bank accounts, a Label bean, which is associated with the IList bean, and a Button bean to switch to AtmPanel when the account is selected (Figure 154).

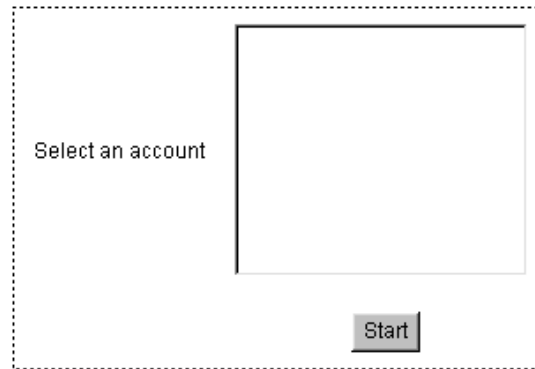


Figure 154. AtmWelcomePanel

You use a GridBagLayout manager to control the placement of your controls when the panel is resized. Figure 155 illustrates the position of the controls and their inset properties. Notice that each control is centered and that the list box spans two columns. Only the list can expand if the panel is stretched out.

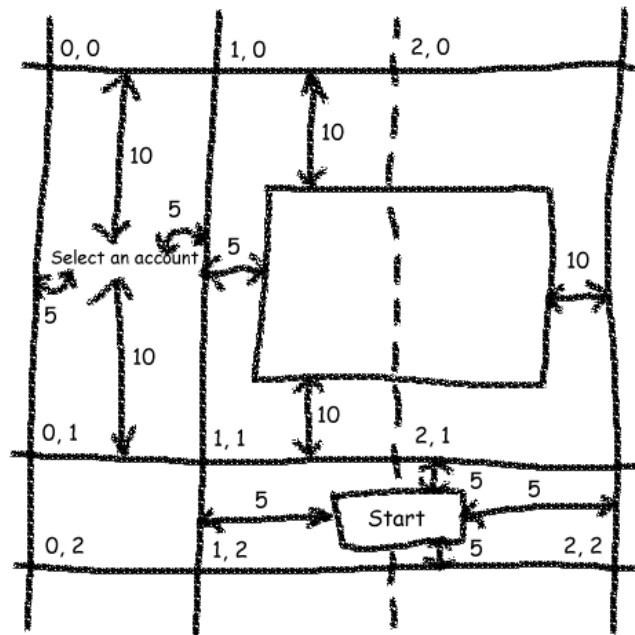


Figure 155. AtmWelcomePanel Layout

To build `AtmWelcomePanel`, follow these steps:

1. In the `ch12` package create an `AtmWelcomePanel` class that inherits from `java.awt.Panel`. Make sure the **Design the class visually** radio button is selected.
2. Click **Finish>** to create the class and load it in the Visual Composition Editor.
3. Double-click the panel to open its property sheet.
4. Change the layout manager to `GridBagLayout`.
5. Drop a Button, a Label, and an IList bean in the panel and change their names to `StartButton`, `SelectLabel`, and `AccountList`, respectively.
6. Double-click `SelectLabel` to display its property sheet and click the button on the right side of the Grid Bag Constraint field to display the Prompter window.
7. Set the label constraints as shown in Figure 155 and Table 10.
8. Repeat Steps 6 and 7 for the remaining controls.
9. Double-click the IList bean to open its property sheet and set the `rows` property to 10 to display 10 accounts vertically.
10. Promote the `actionPerformed(java.awt.event.ActionEvent)` feature of the `StartButton` to make it accessible when `AtmWelcomePanel` is embedded in another composite bean. Set the promoted feature name to `StartButtonAction` to make it simple.

Table 10. `AtmWelcomePanel` `GridBagLayout` Constraints

Property	SelectLabel	AccountList	StartButton
anchor	CENTER	CENTER	CENTER
fill	NONE	BOTH	NONE
gridHeight	1	1	1
gridWidth	1	2	1
gridX	0	1	2
gridY	0	0	1
insets	T10 L5 R5 B10	T10 L5 R10 B10	T10 L5 R5 B10
ipadX	0	0	0
ipadY	0	0	0

Table 10. AtmWelcomPanel GridBagLayout Constraints

Property	SelectLabel	AccountList	StartButton
weightX	0.0	1.0	0.0
weightY	0.0	1.0	0.0

To populate accounts in the AccountList follow these steps:

1. Add a Bank bean on the free-form surface by selecting **Options** → **Add bean...** (or Ctrl+B) from the Visual Composition Editor menu bar.
2. Change its name to *Bank*.
3. Make a property-to-property connection from the *accounts* property of the Bank bean to the *elements* property of the AccountList (see 1 in Figure 156).
4. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

To store the account selected in a variable and transmit it to the other panel, use a BankAccount variable and promote the variable to have it accessible from AtmWelcomePanel:

1. Add a Variable bean on the free-form surface and change its type to *BankAccount*.
2. Change its name to *BankAccount*.
3. Make a property-to-property connection from the *selectedElement* property of the AccountList to the *this* property of the BankAccount variable. In this way you store the selected account in the variable (see 2 in Figure 156).
4. Double-click the property-to-property connection you just built and make sure the Source event drop-down list box has **item** selected. In effect, by default the IList does not generate an event when the currently selected item changes.
5. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

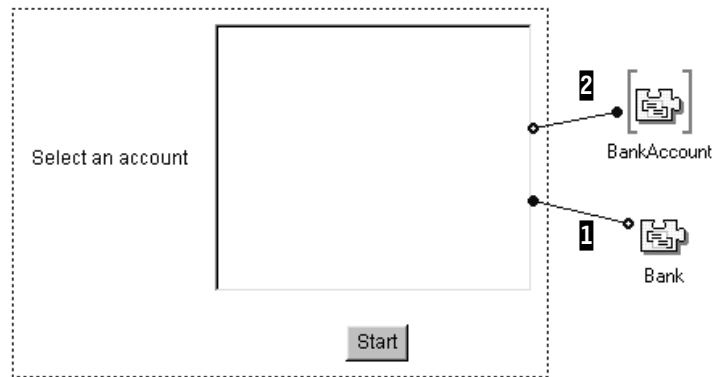


Figure 156. AtmWelcomePanel with Connections

AtmPinPanel

AtmPinPanel allows the user to type in the PIN of the ATM card associated with the selected account. This bean is built up from one Label bean, two Button beans and the AtmKeypadPanel bean (Figure 157). You use a GridBagLayout manager to control the location of the beans when resizing AtmPinPanel.



Figure 157. AtmPinPanel

To build AtmPinPanel, follow these steps:

1. In the ch12 package create an *AtmPinPanel* class that inherits from *java.awt.Panel*. Make sure the **Design the class visually** radio button is selected.

2. Click **Finish>** to create the class and load it in the Visual Composition Editor.
3. Double-click the panel to open its property sheet.
4. Change the layout manager to *GridBagLayout*.
5. Change the background color to *LightGray*.
6. Drop a Label, two Buttons, and an *AtmKeypadPanel* bean in the panel, and change their names to *PinMessageLabel*, *OKButton*, *CancelButton*, and *AtmKeypadPanel*, respectively.
7. Change the text attribute of the Label bean to *Please enter your PIN*, its background color to *White*, and its alignment to *CENTER*.
8. Change the label property of the Button beans to **OK** and **Cancel**.
9. Set the bean layout constraints as shown in Table 11.
10. Promote the *actionPerformed(java.awt.event.ActionEvent)* feature of the *OKButton* and *CancelButton* to make them accessible when *AtmPinPanel* is embedded in another composite bean. Set the promoted feature names to *OKButtonAction* and *CancelButtonAction* to make it simple.
11. Promote the *atmKeypadValue* property from the *AtmKeypadPanel*. Set the promoted feature name to *atmKeypadValue*. This property will be used in *AtmApplet* to check the PIN entered by the user with the *checkPin* method of *AtmCard*.
12. Promote the *clear()* method from the *AtmKeypadPanel*. Set the promoted feature name to *atmKeypadPanelClear()*. This method will be used in *AtmApplet* to reset *ValueTextField* once the PIN entered by the user has been checked by the *checkPin* method of *AtmCard*.
13. Double-click the *AtmKeypadPanel* bean and set its *ValueTextFieldEchoChar* property to *** to avoid echoing the user's PIN. Notice that because you promoted this property when you built *AtmKeypadPanel*, you can access it from its property sheet.
14. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

Table 11. AtmPinPanel GridBagLayout Constraints

Property	PinMes- sageLabel	OKButton	Cancel- Button	AtmKey- padPanel
anchor	CENTER	WEST	EAST	CENTER
fill	NONE	NONE	NONE	BOTH
gridHeight	1	1	1	1
gridWidth	3	1	1	3
gridX	0	0	2	0
gridY	0	2	2	1
insets	T10 L10 R10 B10	T10 L50 R10 B10	T10 L10 R50 B10	T10 L20 R20 B10
ipadX	0	0	0	0
ipadY	0	0	0	0
weightX	0.0	0.0	0.0	1.0
weightY	0.0	0.0	0.0	1.0

AtmPanel

AtmPanel allows the user to create new transactions on the selected account. From this panel deposits and withdrawals can be made. It is also possible to access the AtmHistoryPanel and consult the interest earned by a Savings account for the last transaction.

The panel is a bit more complex than the previous panels because it consists of four different layout managers that take care of the placement of the different controls when the panel is resized. Figure 158 show the AtmPanel layout managers:

- ❑ A BorderLayout manager is used for the AtmPanel itself as the base layout manager (1).
- ❑ In the north region of AtmPanel, a panel displays the current balance of the account. To resize the balance text field, you use a GridBagLayout manager (2).
- ❑ In the center region of AtmPanel, a panel displays a list of four buttons and the AtmPinPanel. The panel organizes these controls with a GridBagLayout manager (3).

- ❑ The four buttons are located in a panel that is associated with a GridLayout manager (4).
- ❑ A panel associated with a FlowLayout manager displays in the south region of the AtmPanel a button to terminate the ATM operations (5).

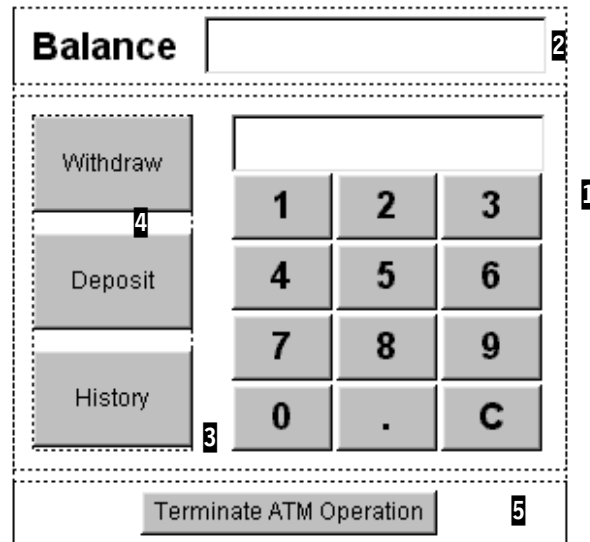


Figure 158. AtmPanel

To build the AtmPanel, follow these steps:

1. In the ch12 package create an *AtmPinPanel* class that inherits from *java.awt.Panel*. Make sure the **Design the class visually** radio button is selected.
2. Click **Finish>** to create the class and load it in the Visual Composition Editor.
3. Double-click the panel to open its property sheet.
4. Change the layout manager to **GridBagLayout**.
5. Select the Containers category and drop a Panel bean on the free-form surface.
6. Change the panel's name to *BalancePanel* and its layout manager to *GridBagLayout*.
7. Drop inside the panel a Label and TextField beans.
8. Change the label text property to *Balance*, its name to *BalanceLabel*, its font size to 18, and the style to *bold*.

9. Change the text field's name to *BalanceTextField*, its editable property to *false*, its column property to *12*, its font size to *18*, and the style to *bold*.
10. Set the constraints for both controls as shown in Table 12.
11. Drag and drop the panel in the *north* region of the *AtmPanel*.
12. Add another panel on the free-form surface.
13. Change the panel's name to *ButtonPanel* and its layout manager to *GridLayout*.
14. Set the *GridLayout* manager properties:
 - columns: *1*
 - hgap: *10*
 - rows: *3*
 - vgap: *10*
15. Add three *Button* beans inside *ButtonPanel* and change their names to *WithdrawButton*, *DepositButton*, and *HistoryButton* and their labels to *Withdraw*, *Deposit*, and *History*, respectively.
16. Promote the *actionPerformed(java.awt.event.ActionEvent)* feature of the *HistoryButton* to make it accessible when *AtmPanel* is embedded in another composite bean. Set the promoted feature names to *HistoryButtonAction* to make it simple.
17. Add another panel to the free-form surface.
18. Change the panel's name to *OperationPanel* and its layout manager to *GridBagLayout*.
19. Drag and drop *ButtonPanel* inside *OperationPanel*.
20. Add an *AtmPinPanel* inside *OperationPanel*.
21. Set the constraints of *ButtonPanel* and *AtmPinPanel* as shown in Table 13. Notice that, because we want to give more space to the *ATMPinPanel* than to the three buttons, we assign bigger weights to *AtmPinPanel* than to *ButtonPanel*.
22. Drag and drop *OperationPanel* in the *center* region of *AtmPanel*.
23. Add one last panel to the free-form surface.
24. Change the panel's name to *TerminatePanel* and its layout manager to *FlowLayout*.
25. Add one *Button* bean inside *TerminatePanel* and change its name to *TerminateButton*.

26. Promote the *actionPerformed(java.awt.event.ActionEvent)* feature of the *TerminateButton* to make it accessible when *AtmPanel* is embedded in another composite bean. Set the promoted feature name to *TerminateButtonAction* to make it simple.
27. Drag and drop *TerminatePanel* in the **south** region of *AtmPanel*.
28. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

Table 12. BalancePanel GridBagLayout Constraints

Property	BalanceLabel	BalanceTextField
anchor	WEST	EAST
fill	NONE	HORIZONTAL
gridHeight	1	1
gridWidth	1	1
gridX	0	1
gridY	0	0
insets	T5 L10 R0 B5	T5 L0 R10 B5
ipadX	0	0
ipadY	0	0
weightX	0.0	1.0
weightY	0.0	0.0

Table 13. OperationPanel GridBagLayout Constraints

Property	ButtonPanel	AtmPinPanel
anchor	WEST	EAST
fill	BOTH	BOTH
gridHeight	1	1
gridWidth	1	1
gridX	0	0

Table 13. OperationPanel GridBagLayout Constraints

Property	ButtonPanel	AtmPinPanel
gridY	0	1
insets	T10 L10 R10 B10	T10 L10 R10 B10
ipadX	0	0
ipadY	0	0
weightX	0.5	2.0
weightY	0.5	2.0

Once you have tested your panel, you need to add a *BankAccount* variable to interact with the account selected from *AtmWelcomePanel*. From this variable you can make a withdrawal or a deposit:

1. Add a Variable bean to the free-form surface and change its type to *BankAccount*.
2. Change its name to *BankAccount*.
3. Make a property-to-property connection from the *balance* property of the *BankAccount* variable to the *text* property of the *BalanceTextField*. In this way you update the text field with the current balance of the account (see connection 1 in Figure 159). When you create this connection you get a warning message: “Source and target property types, double and java.lang.String, are not compatible. Do you want to continue?” because the balance property is of type double whereas the text property is of type String. Ignore this message and click **OK**. The Visual Composition Editor will generate the proper conversion code for you. Notice that, because balance is a bound property feature, it automatically triggers an event each time it is updated. This event is set as the source event in the Source event drop-down list box of the connection property sheet.
4. Promote the *this* property feature of the *BankAccount* variable to make it accessible when you embed *AtmKeypadPanel* in another composite bean.
5. Connect the *actionPerformed(java.awt.event.ActionEvent)* event from the *WithdrawButton* to the *debit(double)* method of the *BankAccount* variable (see connection 2 in Figure 159).
6. Connect the *atmKeypadValue* property from the *AtmKeypadPanel* to the *anAmount* parameter of the previous connection (see connection 3 in Figure 159).

7. Connect the `actionPerformed(java.awt.event.ActionEvent)` event from the DepositButton to the `credit(double)` method of the BankAccount variable (see connection 4 in Figure 159).
8. Connect the `atmKeypadValue` property from the AtmKeypadPanel to the `anAmount` parameter of the previous connection (see connection 5 in Figure 159).
9. Connect the `actionPerformed(java.awt.event.ActionEvent)` event from the WithdrawButton to the `clear()` method of the AtmKeypadPanel to clear ValueTextField once an operation has been completed (see connection 6 in Figure 159).
10. Connect the `actionPerformed(java.awt.event.ActionEvent)` event from the DepositButton to the `clear()` method of the AtmKeypadPanel to clear ValueTextField once an operation has been completed (see connection 7 in Figure 159).
11. Save and test the bean by clicking the left-most icon of the Visual Composition Editor.

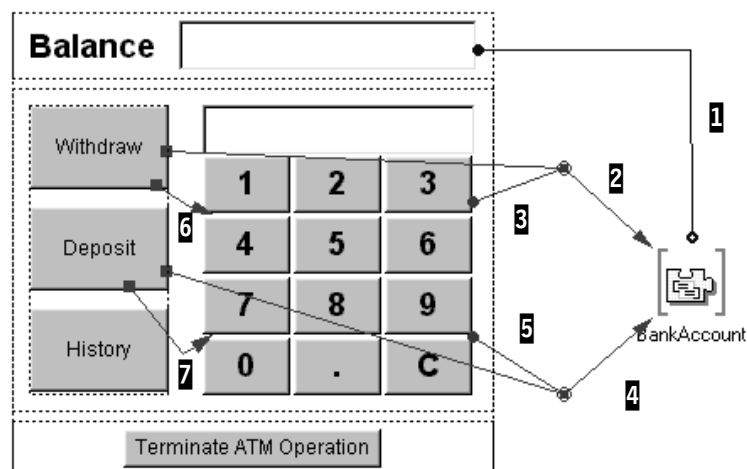


Figure 159. AtmPanel with Connections

The two connections starting from WithdrawButton and DepositButton are executed in sequence. To check the sequence in which they are fired, you can access the **Reorder Connections From...** option from the pop-up menu of either button. Figure 160 shows the order of the connections that are fired from the Withdrawal button.

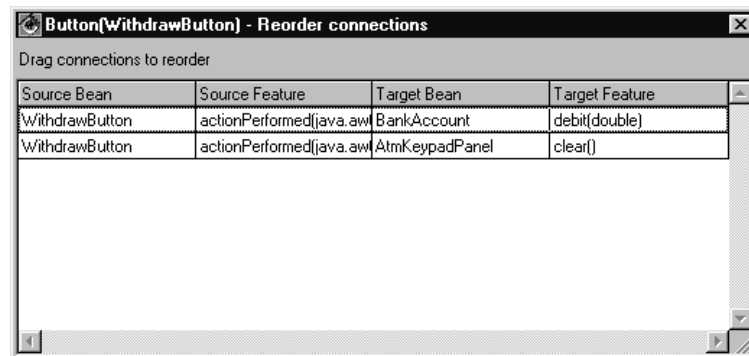
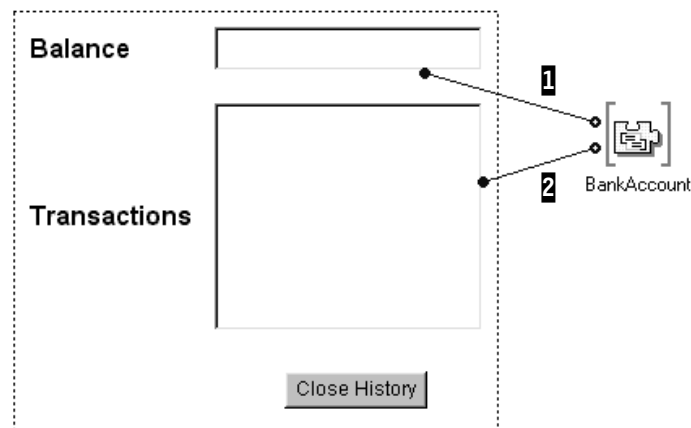


Figure 160. Reorder Connections Window

AtmHistoryPanel

Now that you have built *AtmPanel*, you are going to find *AtmHistoryPanel* very simple to build! *AtmHistoryPanel* displays the account balance and lists the latest transactions (Figure 161).

Figure 161. *AtmHistoryPanel*

To build the *AtmHistoryPanel*, follow these steps:

1. In the `ch12` package create an *AtmHistoryPanel* class that inherits from *java.awt.Panel*. Make sure the **Design the class visually** radio button is selected.
2. Click **Finish>** to create the class and load it in the Visual Composition Editor.
3. Double-click the panel to open its property sheet.

4. Change the layout manager to *GridBagLayout*.
5. Drop two Label beans and change their label properties to *Balance* and *Transactions* as shown in Figure 161 and their names to *BalanceLabel* and *TransactionLabel*, respectively.
6. Add one TextField bean to *AtmHistoryPanel* and change its label to *Close History* and its name to *CloseButton*.
7. Promote the *actionPerformed(java.awt.event.ActionEvent)* feature of the *CloseButton* to make it accessible when *AtmHistoryPanel* is embedded in another composite bean. Set the promoted feature name to *CloseButtonAction* to make it simple.
8. Add an *IList* bean to *AtmHistoryPanel* and change its name to *TransactionList*.
9. Set the constraints for each bean as shown in Table 14.

Table 14. *AtmHistoryPanel* *GridBagLayout* Constraints

Property	Balance-Label	TransactionLabel	Balance-TextField	TransactionList
anchor	WEST	WEST	EAST	EAST
fill	NONE	NONE	HORIZONTAL	BOTH
gridHeight	1	1	1	1
gridWidth	1	1	1	1
gridX	0	0	1	1
gridY	0	1	0	2
insets	T10 L10 R0 B10	T10 L10 R0 B10	T10 L0 R10 B10	T10 L0 R10 B10
ipadX	0	0	0	0
ipadY	0	0	0	0
weightX	0.0	0.0	0.5	1.0
weightY	0.0	0.0	0.5	1.0

To populate *BalanceTextField* and *TransactionList* with the information from the select account, you have to add a *Variable* bean to the free-form surface and make the associated property-to-property connections:



1. Add a Variable bean of type `BankAccount` to the free-form surface.
2. Change the variable name to *BankAccount*.
3. Create a property-to-property connection from the *balance* property of the `BankAccount` variable to the *text* property of `BalanceTextField` (see 1 in Figure 161).
4. Create a property-to-property connection from the *log* property of the `BankAccount` variable to the *elements* property of `TransactionList` (see 2 in Figure 161).
5. Promote the *this* property feature of the variable to transmit the account selected from `AtmWelcomePanel`.
6. Save `AtmHistoryPanel`.

Congratulations! You have built all of the pieces you need to assemble your applet.

Putting Your Applet Together

In this section you complete your applet by assembling the panels you built in the previous sections. Variable beans are used to transmit the account selected from one panel to another.

The ATM applet that derives from the `Panel` class is provided with a `CardLayout` manager which enables the user to switch from one panel to another by using the different buttons available in each panel. To build the applet you first assemble the different panel in the main panel, using the `CardLayout` manager. Then you create a custom event in the `BankAccount` bean that triggers access to `AtmPanel` if the user enters the correct PIN in `AtmPinPanel`.

Assembling the ATM Panels

To create your ATM applet, follow these steps:

1. Create an applet, *AtmApplet*, in package *chap12* in the *Learn Java* project. Make sure the **Design your applet visually** radio button is selected before clicking **Finish>**. The Visual Composition Editor opens with the `AtmApplet` panel on the free-form surface.
2. Change the layout property of the `AtmApplet` to *BorderLayout* and accept the default settings.
3. Add an `AtmAccountPanel` bean in the *north* region of `AtmApplet` and change its name to *AtmAccountPanel*.

4. Add a Panel bean to the free-form surface, change its name to *MainPanel*, and change its layout property to *CardLayout* (the Panel bean is located in the *Containers* category). This is the panel that will hold the different panels of the ATM, one panel per page.
5. Add an *AtmWelcomePanel* bean inside *MainPanel*. Notice that *AtmWelcomePanel* is a truly living bean and that, when you drop it in *MainPanel*, its *AccountList* is filled up by the *Bank* bean embedded in *AtmWelcomePanel*.
6. Add an *AtmPinPanel* bean on top of the *AtmWelcomePanel* bean you just dropped. Change its name to *AtmWelcomePanel*. Notice that the *AtmWelcomePanel* bean disappears underneath the *AtmPinPanel* bean and that you can now switch from one panel to the other by using the **Switch to** option from the *MainPanel* pop-up menu (Figure 162).
7. Add an *AtmPanel* bean on top of the *AtmPinPanel* bean. Change its name to *AtmPanel*.
8. Add an *AtmHistoryPanel* bean on top of the *AtmPanel* bean. Change its name to *AtmHistoryPanel*.
9. Once your *MainPanel* contains all four ATM panels, drag and drop it in the center region of *AtmApplet*. Notice that the *AccountPanel* bean is always displayed no matter which panel is selected in *MainPanel*. This is the interface look and feel described in Figures 133 through 136.
10. Save and test your applet by clicking the toolbar **Test** button.

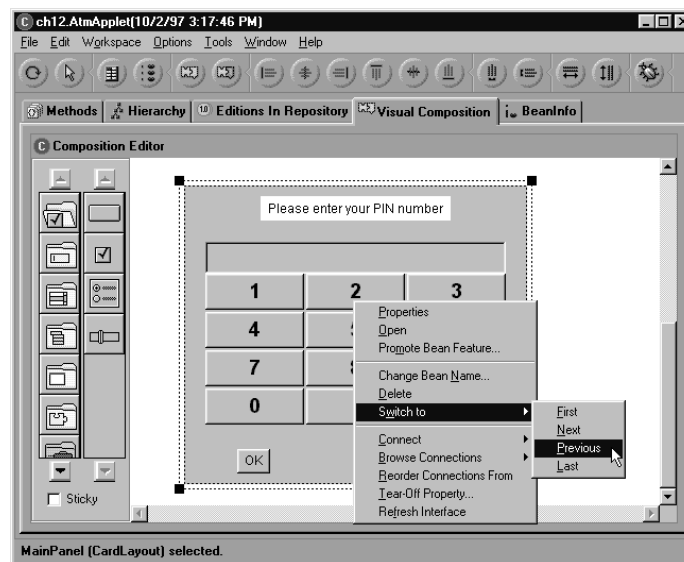


Figure 162. Switching Page with a CardLayout Manager

When you test your applet, there is not much you can do because you cannot switch from `AtmWelcomePanel` to `AtmPinPanel`. In addition, when you select an account in `AccountList`, the account information is not populated in the `TextField` beans of `AtmAccountPanel`. Do not panic! Everything is ready for your ATM to breathe some life. You just need a couple of `Variable` beans and connections to make that happen.

Switching Cards from AtmWelcomePanel

To switch from the `AtmWelcomePanel` card to the `AtmPinPanel` card in `MainPanel`, call the *next* method of the `CardLayout` class when the user clicks the **Start** button of `AtmWelcomePanel`. Follow these steps:

1. Add a `Variable` bean to the free-form surface and change its type to `CardLayout`. Change its name to `CardLayout`. This variable will hold the layout manager of `MainPanel`. In this way you can call the *next* method when the user clicks the **Start** button.
2. Connect the *layout* property of `MainPanel` to the *this* property of the variable (see 1 in Figure 163). Your variable is a torn-off property feature of the `MainPanel` layout property.
3. Connect the *startButton.Action(java.util.EventObject)* event, which has been promoted in `AtmWelcomePanel`, to the *next(java.awt.Container)* method of the `CardLayout` variable (see 2 in Figure 163). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.
4. Connect the *this* property of `MainPanel` to the *parent* parameter of the connection you just made (see 3 in Figure 163). The connection should turn plain.
5. Save and test your applet by clicking the toolbar **Test** button. Notice that you can switch to `AtmPinPanel` when you click the **Start** button.

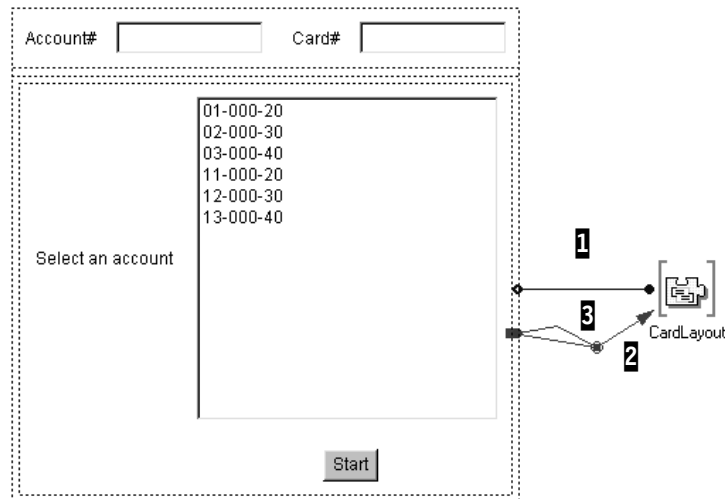


Figure 163. Using Connections to Switch Cards

Switching Cards from *AtmPinPanel*

From the *AtmPinPanel* card, the user can switch to the *AtmHistoryPanel* card by entering a valid PIN and clicking **OK**. (However, to test the applet immediately, you do not need to implement the mechanism that checks the PIN at this time.) In addition, the user can switch back to the *AtmWelcomePanel* card by clicking the **Cancel** button. To implement this behavior:

1. Switch to the *AtmPinPanel* card.
2. Connect the `oKButton.Action(java.util.EventObject)` event, which has been promoted in *AtmPinPanel*, to the `next(java.awt.Container)` method of the *CardLayout* variable (see 1 in Figure 164). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.
3. Connect the `this` property of *MainPanel* to the `parent` parameter of the connection you just made (see 2 in Figure 164). The connection should turn plain.
4. Connect the `cancelButton.Action(java.util.EventObject)` event, which has been promoted in *AtmPinPanel*, to the `previous(java.awt.Container)` method of the *CardLayout* variable (see 3 in Figure 164). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.

5. Connect the *this* property of MainPanel to the *parent* parameter of the connection you just made (see 4 in Figure 164). The connection should turn plain.
6. Save and test your applet by clicking the toolbar **Test** button.

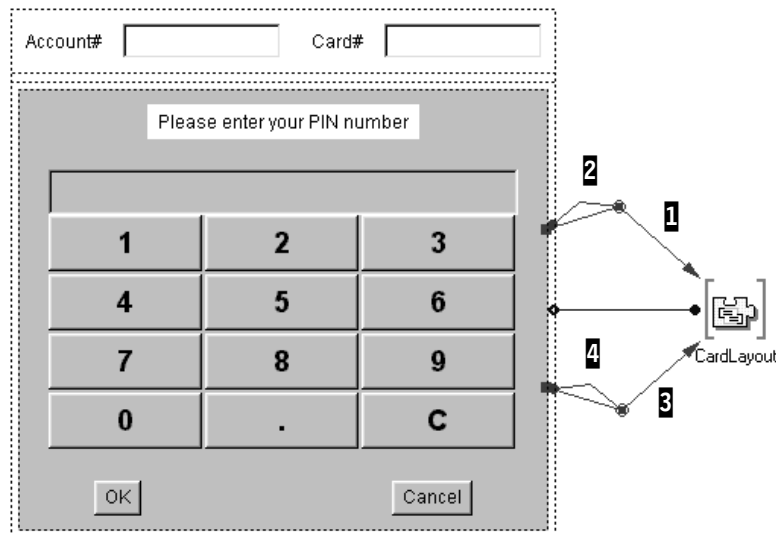


Figure 164. Switching Cards from AtmPinPanel

Switching Cards from AtmPanel

From the AtmPanel card the user can switch to the AtmHistoryPanel card by clicking the **History** button or switch back to the AtmWelcomePanel card by clicking the **Terminate ATM Operation** button. To implement this behavior:

1. Switch to the AtmPanel card.
2. Connect the `historyButton.Action(java.util.EventObject)` event, which has been promoted in AtmPanel, to the `next(java.awt.Container)` method of the CardLayout variable (see 1 in Figure 165). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which switch.
3. Connect the *this* property of MainPanel to the *parent* parameter of the connection you just made (see 2 in Figure 165). The connection should turn plain.
4. Connect the `terminateButton.Action(java.util.EventObject)` event, which has been promoted in AtmPanel, to the `first(java.awt.Container)` method of the CardLayout variable

(see **3** in Figure 165). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.

5. Connect the *this* property of MainPanel to the *parent* parameter of the connection you just made (see **4** in Figure 165). The connection should turn plain.
6. Save and test your applet by clicking the toolbar **Test** button.

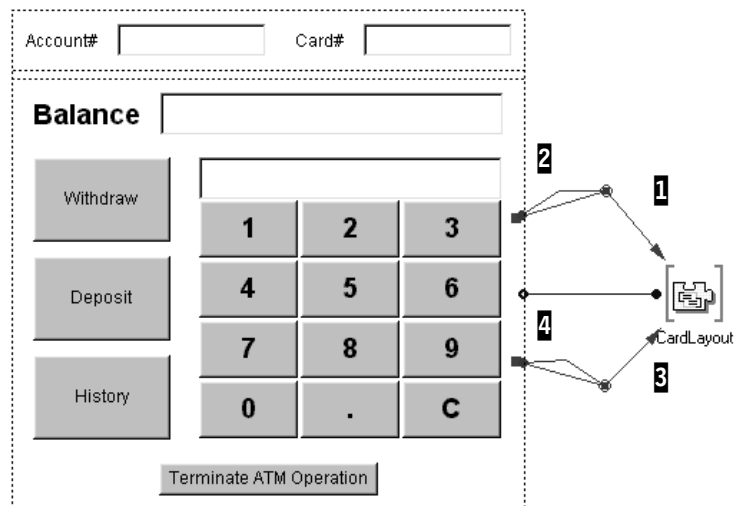


Figure 165. Switching Cards from AtmPanel

Switching Cards from AtmHistoryPanel

From the AtmHistoryPanel card, the user can only switch back to the AtmPanel card by clicking the **Close History** button. To implement this behavior:

1. Switch to the AtmHistoryPanel card.
2. Connect the *closeButton.Action(java.util.EventObject)* event, which has been promoted in AtmHistoryPanel, to the *previous(java.awt.Container)* method of the CardLayout variable (see **1** in Figure 165). Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.
3. Connect the *this* property of MainPanel to the *parent* parameter of the connection you just made (see **2** in Figure 165). The connection should turn plain.

4. Save and test your applet by clicking the toolbar **Test** button. The dynamic of the applet should be fully implemented, and you should be able to switch back and forth between the different panels.

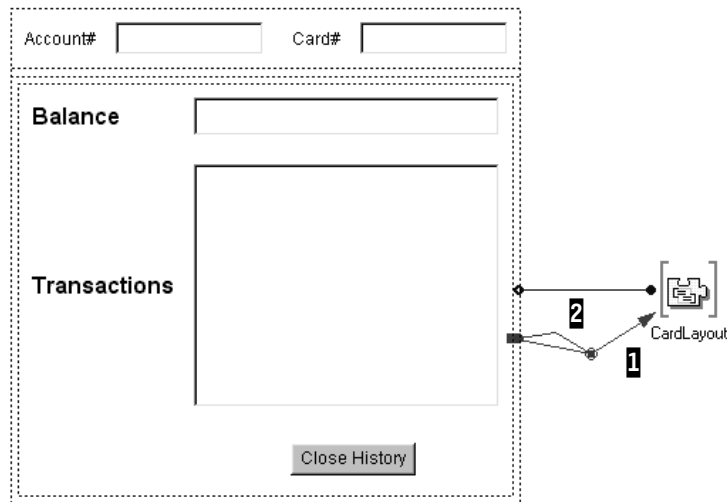


Figure 166. Switching Cards from AtmHistoryPanel

Transmitting the BankAccount Object

Once the user has selected an account from the account list of the AtmWelcomePanel card, that account must be transmitted across every other panel by using the internal BankAccount variable that you have promoted.

1. Switch to the AtmWelcomePanel card.
2. Add a BankAccount variable to the free-form surface.
3. Connect the *this* property of the BankAccount variable to the *bankAccountThis* promoted property of the AtmAccountPanel bean. In this way, when the user selects an account, its information is propagated to the AtmAccountPanel text fields. Notice that you could also build the same connection by tearing off the bankAccountThis property feature that you promoted in AtmWelcomePanel.
4. Switch to the AtmPanel card.
5. Connect the *this* property of the BankAccount variable to the *bankAccountThis* promoted property of the AtmPanel bean. In this way, when the user creates transactions, those transac-

tions that apply to the selected account and the account balance are displayed in `TransactionList` and `BalanceTextField`, respectively.

6. Switch to the `AtmHistoryPanel` card.
7. Connect the *this* property of the `BankAccount` variable to the *bankAccountThis* promoted property of the `AtmHistoryPanel` bean. In this way, the account transactions are displayed in `TransactionList`.
8. Save and test your applet.

Congratulations! Your applet is almost complete. The user should be able to credit and debit a selected account. The account balance should be updated accordingly in the `AtmPanel` card, and the transactions should be displayed in the `AtmHistoryPanel` card.

Notice, however, that if you try to withdraw money from an account that has an insufficient balance, you will not get any feedback from the ATM. You can improve your applet by displaying the `InsufficientFundsException` that you created in “Defining the `InsufficientFundsException` Class” on page 150.

Displaying a Java Exception in a Message Box

VisualAge for Java provides you with a useful `IMessageBox` bean that you can use with the Visual Composition Editor to display any message. (Although the `IMessageBox` bean is not available in the trial version, it is included in the CD-ROM in the `COM\ibm\ivj\javabeans` directory. Refer to Appendix A, “VisualAge for Java Installation,” on page 441, for loading this bean in your workspace).

Predefined and user-defined exceptions can be caught from the connection that fires the method that raises the exception. You just connect the *exceptionOccurred* event feature, which shows up in the connection pop-up menu, to any method you want to execute. In our case, you connect this event to the `showException` method of a `IMessageBox` bean:

1. Open the `AtmPanel` bean in the Visual Composition Editor.
2. Select **Options->Add Bean...** from the menu bar and add an *IMessageBox* bean. This bean is located in the `COM.ibm.ivj.javabeans` package that must be loaded in your workspace.
3. Drop the `IMessageBox` bean on the free-form surface and change its name to *MessageBox* and its title to *Withdrawal Exception*.

4. Connect the *exceptionOccurred* event from the connection that fires the *debit* method of *BankAccount* (see connection 1 in Figure 167) to the *showException* method of *MessageBox* (see connection 2 in Figure 167). The connection appears as a dotted line because you must provide a *java.lang.Throwable* object to the *showException* method.
5. Double-click the connection you just created and check the **Pass event data** checkbox to pass the event and the exception to the connection (see connection 3 in Figure 167). The connection should be plain now.
6. Save *AtmPanel*.

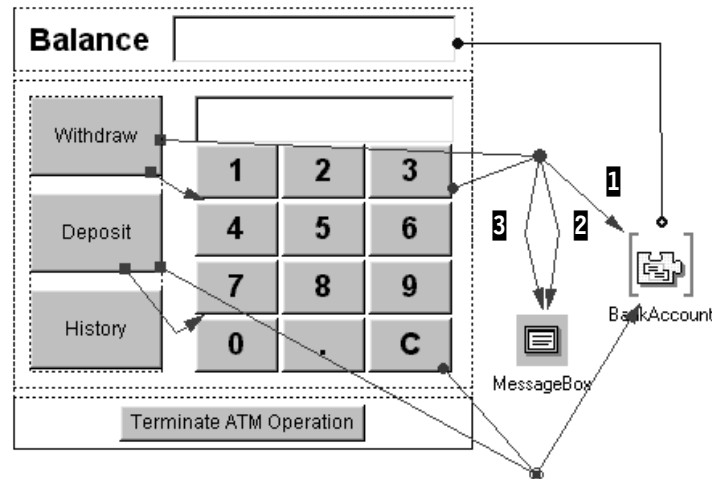


Figure 167. Displaying a Java Exception in an *IMessageBox* Bean

When you test your *AtmApplet* and try to withdraw an amount that is greater than the current balance, a message box should pop up with the *InsufficientFundsException* message (Figure 168).

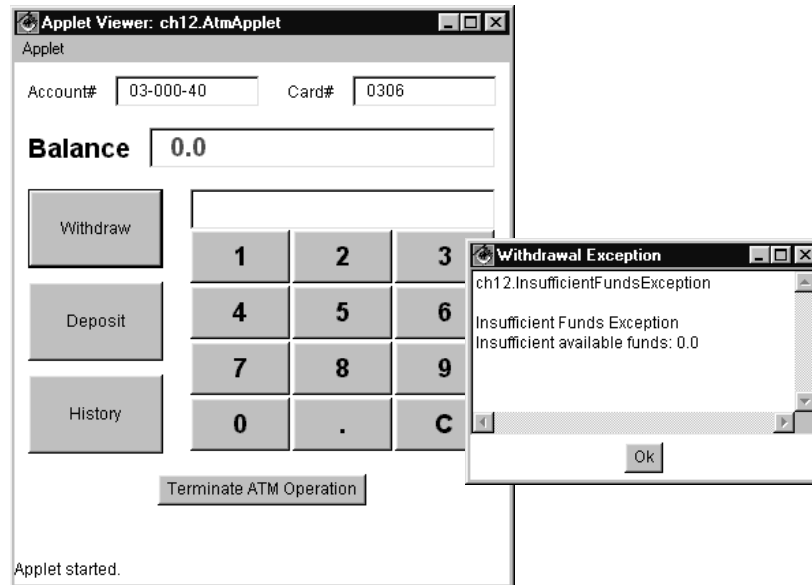


Figure 168. `InsufficientFundsException` Displayed in a Message Box

Your applet is almost complete. A little detail (which is actually very important for the bank!) must be implemented to ensure that the user accesses the `AtmPanel` card only if he or she provides a valid PIN in the `AtmPinPanel` card.

Creating a Custom Event

When the user enters the ATM card PIN in the `AtmPinPanel` card and clicks the **OK** button, the PIN is sent to the `BankAccount` variable that holds the selected account. The `BankAccount` variable delegates the checking of the PIN to the `AtmCard` bean, which runs the `checkPin` method you created in “Creating the `AtmCard` Class” on page 279. Only the `AtmCard` bean knows how to check the PIN.

To call the `checkPin` method of `AtmCard`, you implement a `checkPin` method in the `BankAccount` bean, which calls the `checkPin` method of its associated `AtmCard` object.

If the PIN entered by the user is valid, the `checkPin` method of the `BankAccount` bean generates a `PinCheckedOkEvent`. This event is used to fire an event-to-method connection that switches the `MainPanel` to the `AtmPanel` card.

If the PIN entered by the user is not valid, the `checkPin` method of the `BankAccount` does not generate an event, and the connection is not fired. The user is stuck on the `AtmPinPanel` page and cannot access the account.

Several agents are involved in this scenario:

- ❑ **Event object:** You create a `PinCheckedOkEvent` class, an instance of which is fired by the `checkPin` method of the `BankAccount` if the `AtmCard` validates the PIN entered by the user.
- ❑ **Event source:** You extend `BankAccount` by adding the `checkPin` method so that it becomes an event source that fires the `PinCheckedOkEvent` when the PIN is checked OK by the `ATMCard` `checkPin` method.
- ❑ **Event listener interface:** You create your own event listener interface which extends the `EventListener` interface. Your interface has one method, `handlePinCheckedOk`. This method will be implemented by any event listener that implements your interface.
- ❑ **Event listener:** By making a connection from the `PinCheckedOkEvent` generated by `BankAccount` to the `next(java.awt.Container)` method of `MainPanel`, you make `MainPanel` an event listener. This event listener implements the `handlePinCheckedOk` method by calling the `next(java.awt.Container)` method to switch to the `AtmPanel` card.

Figure 169 represents an event-trace diagram of the scenario. The user clicks the `AtmPinPanel` OK button to generate an `ActionEvent`. This event fires an event-to-method connection, which calls the `checkPin` method of the `BankAccount` object. The `checkPin` method delegates the PIN validation by calling the `AtmCard` `checkPin` method. If the PIN is correct, the `checkPin` method of the `AtmCard` returns a true value. This boolean value is checked by the `checkPin` method of `BankAccount`, which, in this case, sends a `PinCheckedOkEvent`. This event fires an event-to-method connection, which calls the `next` method of the `MainPanel` to switch to the `AtmPanel` card.

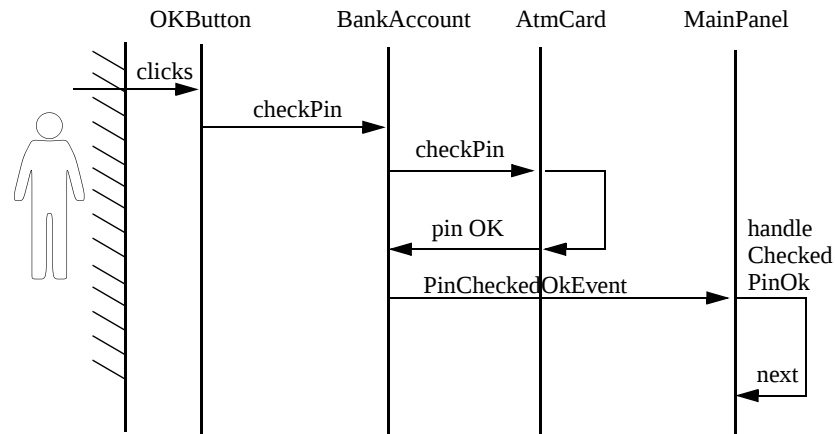


Figure 169. Check Pin Number Event-Trace Diagram

Now that you have the big picture in mind, let's implement it!

Creating a New Listener Interface Feature

First, create a new listener interface and its associated event class:

1. Open the class browser with `ch12.BankAccount`
2. Switch to the BeanInfo page.
3. From the Features pull-down menu, select **New Listener Interface...** to create a listener interface and its associated event.
4. The New event Listener SmartGuide opens.
5. Type `pinCheckedOk` as the Event Name. The remaining fields are completed automatically according to the syntax pattern of the JavaBeans specification (Figure 170).
6. Switch to the next page by clicking the **Next>** button.
7. The Event Listener Methods SmartGuide is displayed and prompts you for a method name (Figure 171). This method name is the method that any listener that wants to support the interface must implement. In your case enter `handlePinCheckedOkEvent` and click the **Add** button to add the method to the Event Listener Methods list.
8. Click **Finish>** to create the event class, the listener interface, and the associated methods.

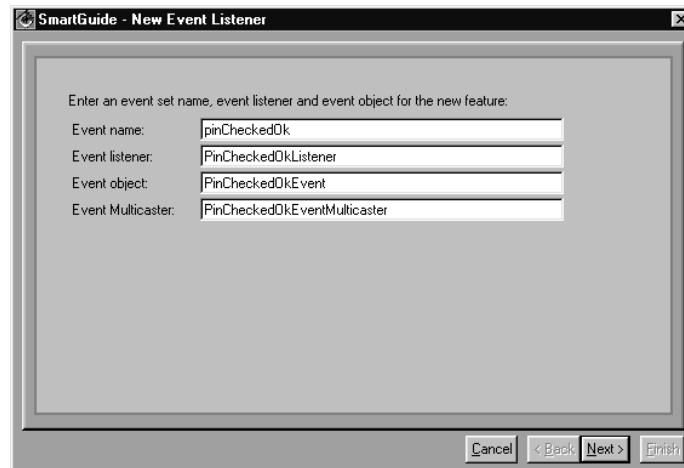


Figure 170. SmartGuide - New Event Listener Window

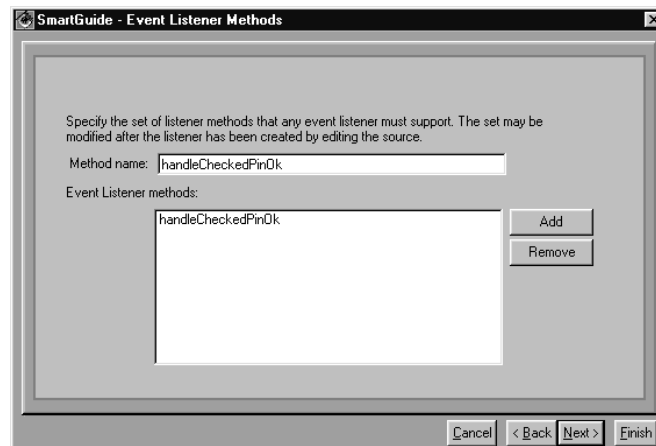


Figure 171. SmartGuide - Event Listener Methods Window

If you switch to the Methods page, you notice that several methods have been generated by the Visual Composition Editor to support your new interface. The `addPinCheckedOkListener` and `removePinCheckedOkListener` methods manage the list of listeners that is maintained by `BankAccount` and that contains all objects that register with `BankAccount` to be notified by `PinCheckedOkEvent`. The `fireHandlePinCheckedOk` is the method you use to fire the `PinCheckedOkEvent`, as you will see in “Making `BankAccount` an Event Source” on page 342. In addition, a `PinCheckedOkListener` interface has been created with the `handlePinCheckedOk` method, which has to be implemented by any listener that registers with `BankAccount` to receive a `PinCheckedOkEvent` object.

Making BankAccount an Event Source

The next step is to add to `BankAccount` a `checkPin` method that calls the `AtmCard` `checkPin` and fires a `PinCheckedOkEvent`, depending on whether the PIN has been validated as correct by the `AtmCard` class.

1. From the Features pull-down menu, select **New Method Feature...** to create a new *public* method in `BankAccount`.
2. Enter *checkPin* as the method name, *boolean* as the return type, and *1* as the parameter count. Click **Next>**.
3. On the Parameter 1 page, enter *aPin* as the parameter name and *java.lang.String* as the parameter type. Click **Finish>** to create the method.
4. Select the *checkPin* method in the Definitions pane of the BeanInfo page. The source code is displayed in the bottom Source pane.
5. Modify the `checkPin` method:

```
public boolean checkPin(String aPin) {
    boolean correct = getAtmCard().checkPin(aPin);
    if (correct) {
        fireHandlePinCheckedOk(new PinCheckedOkEvent(this));
    }
    return correct;
}
```

6. Save the `BankAccount` bean.

Notice that the `checkPin` method fires a `PinCheckedOkEvent` object by using the `fireHandlePinCheckedOk`. This method was generated by the Visual Composition Editor when you created the `PinCheckedOkListener` listener interface.

Making the Connections

To complete your applet, you just make a few connections to activate the PIN checking scenario. Open the Visual Composition Editor on your `AtmApplet` and follow these steps:

1. Select `MainPanel` and switch to the `AtmPinPanel` card.
2. Remove the event-to-method connection from the **OK** button to the `CardLayout` variable to prevent the user from switching automatically to the `AtmPanel` card without checking the PIN.

3. Connect the *oKButton.Action(java.util.EventObject)* event, which has been promoted in *AtmPinPanel*, to the *checkPin(java.lang.String)* method of the *BankAccount* variable (see 1 in Figure 172). Notice that the connection is dotted because it needs a parameter: the PIN to check.
4. Connect the *atmKeypadValue* property of *MainPanel*, which has been promoted in *AtmPinPanel* to the *aPin* parameter of the connection you just made (see 2 in Figure 172). The connection should turn plain.
5. Connect the *pinCheckedOk* event of the *BankAccount* variable to the *next(java.awt.Container)* method of the *CardLayout* variable. When *BankAccount* fires *pinCheckedOk*, it will make *MainPanel* switch to the *AtmPanel* card. Notice that the connection is dotted because it needs a parameter: the parent container of the next card to which to switch.
6. Connect the *this* property of *MainPanel* to the *parent* parameter of the connection you just made (see 4 in Figure 172). The connection should turn plain.
7. Connect the *oKButton.Action(java.util.EventObject)* event, which has been promoted in *AtmPinPanel*, to the *clear()* method of the *AtmPinPanel* that has been promoted (see 5 in Figure 172). This connection clears *ValueTextField* once a PIN has been entered and processed.
8. Connect the *cancelButton.Action(java.util.EventObject)* event, which has been promoted in *AtmPinPanel*, to the *clear()* method of the *AtmPinPanel* that has been promoted (see 6 in Figure 172). This connection clears *ValueTextField* once a PIN has been entered and processed.
9. Save and test your applet. The user should not be able to access the *AtmPanel* card until he or she has entered the correct PIN of the account selected. Remember that the PIN is calculated from the account identifier. It consists of the first two digits of the account identifier followed by 123.

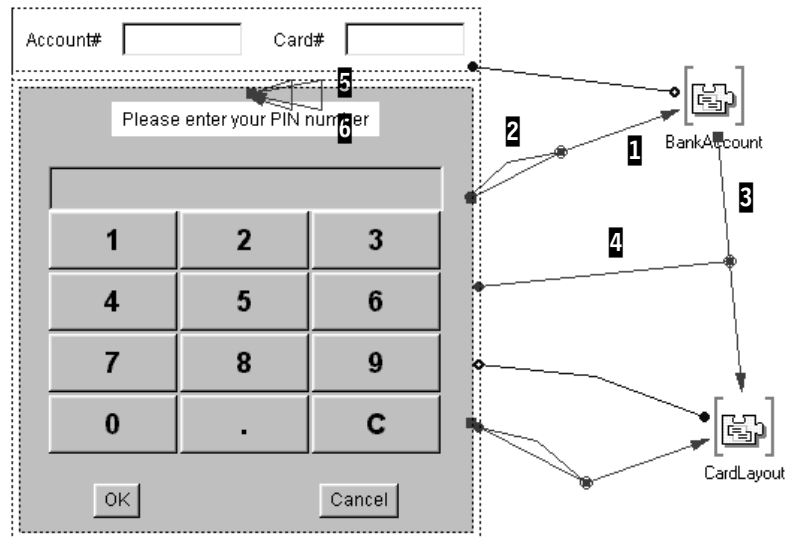
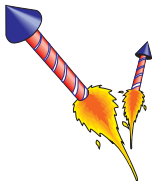


Figure 172. Enabling PIN Checking with Connections



You have now completed your ATM applet! Notice that the logic of the banking model you developed takes place mostly in `AtmPanel` where the transactions occur. Notice also that the applet is well layered and that few connections appear at each layer.

In the next section we show you how you can convert your applet to an application.

Running Your ATM Applet As an Application

With the Visual Composition Editor you can run your applet easily as an application. You can also provide your applet with a frame window and convert it to a Java application. Once your applet is converted, you can add menus to its frame window.

Converting Your Applet

When you create your `AtmApplet` visually, the Visual Composition Editor creates a main method, which enables you to run your applet as an application. This main method creates a `Frame` object and encloses your applet in the frame.

To run your applet as an application, from the Workbench select `AtmApplet` and click the runner icon in the toolbar. A dialog is then displayed where you click the **Run main()** button as shown in Figure 173.

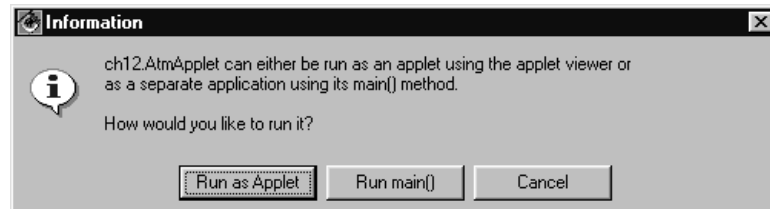


Figure 173. Running `AtmApplet` As an Application

If you prefer you can create a Java application and embed your applet in it. In this way you can visually provide your application with menus:

1. Create in the `ch12` package an `AtmApplication` class that derives from `java.awt.Frame`. Choose to design the application visually.
2. Double-click the frame window and change its *layout* property to *BorderLayout*.
3. Add your `AtmApplet` in the *center* region of the `AtmApplication` client area.
4. Test and save your application by clicking the **Test** icon from the toolbar.

Voila! You have created an application from your applet using the Visual Composition Editor. Now you can visually add menus to your application as described in “Adding Menus to Your Application” on page 270.

The Final Touch

To complete your applet, you are going to enable the user to access the calculator you built in Chapter 11. You add an additional `Button` bean to the `AtmPanel` to enable the user to start the calculator.

Using Factory Bean

Get methods are associated with each bean you drop on the free-form surface of your application or applet. Those get methods create on the heap an instance of a bean the first time they are called. When your program starts, it calls the `initialize` method, which in turn calls the different get methods to create the beans that make up your applet or application.

With the Visual Composition Editor, you can also create bean instances at run time by using a Factory bean. Factory beans enable your application or applet to dynamically create visual or nonvisual beans. Factory beans differ from beans that are added to the free-form surface and created statically when the program starts.

Like Variable beans, Factory beans are placeholders for other beans. Each Factory bean must be set to the type of the class it represents. The Factory bean works in tandem with a Variable bean which represents the instance created. You must use Variable beans with Factory beans each time you access the properties or activate the methods of the instance created.

To run the `CalculApp` application from your applet, you provide the `AtmPanel` bean with a `Button` bean, which triggers the new method of a Factory bean of type `CalculApp`. Once the `CalculApp` instance is created by the Factory bean, you just call the `show` method of the `CalculApp` to display it to the user.

Running CalculApp from AtmApplet

To modify your `AtmPanel` and run `CalculApp` dynamically, follow these steps:

1. Open `AtmPanel` in the Visual Composition Editor.
2. Add a `Button` bean in the `ButtonPanel` bean. The `ButtonPanel` bean resizes automatically to display the four buttons in two rows.
3. Double-click the `ButtonPanel` bean and change its `GridLayout` constraint to *4 rows* to display the four buttons vertically in one column.
4. Change the `Button` bean name to *CalculatorButton* and its label to *Calculator*.
5. Select the *Models* category in the palette and drop a *Factory* bean on the free-form surface (the Factory bean is the second bean in the category).

6. Change the Factory bean name to *CalculAppFactory*.
7. From the Factory bean pop-up menu, select **Change Type....**. A dialog is displayed to let you choose the type of instance you want the Factory bean to create.
8. Choose *ch11.CalculApp* as the type and click **OK** to close the dialog.
9. Connect the *actionPerformed(java.awt.event.ActionEvent)* event from *CaculatorButton* to the *CalculApp()* method of *CalculAppFactory*. Notice that you can also connect to the *CalculApp(java.lang.String)* constructor, which takes the *CalculApp* title window *String* as a parameter.
10. Connect the *actionPerformed(java.awt.event.ActionEvent)* event from *CaculatorButton* to the *show()* method of *CalculAppFactory*.
11. Save *AtmPanel*.

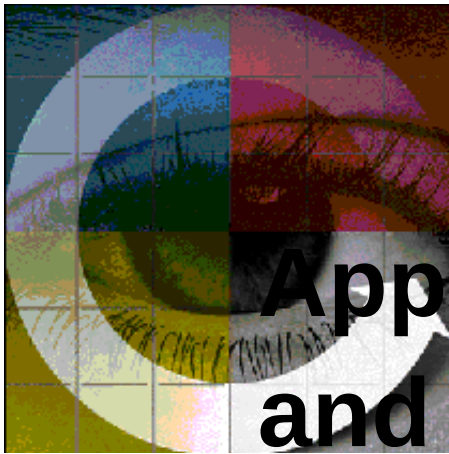
Good job! You can now fully enjoy your ATM applet and start the calculator application from *AtmPanel* in a separate window.

Summary



Using the BeanInfo page, you can easily make your model classes behave as fully functional beans. Once you have designed and tested the view of your applet or application, using the Visual Composition Editor, you can incorporate your model beans on the free-form surface and complete your program by connecting your beans together.

Part 3



Applet Publishing and Advanced Features

Now that your ATM applet is completed, you must feel more confident with the IDE and the Visual Composition Editor. In this part, we show you how you can improve your applet by providing multithreading, national language, and run-time type information support. Then, we detail the process of publishing your applet on the Web and explain how to set up your Web server. Finally, we introduce you to a new set of tools that come with the VisualAge for Java Enterprise Edition. You can use those tools to access data that resides in a relational database, CICS transactions, or distributed Java objects. We also tell you how you can use VisualAge for Java tools to distribute your objects in a network. We conclude by sharing with you some thoughts and ideas that we developed to improve the ATM applet.

13

Improving Your ATM Applet

In this chapter you improve your ATM applet by providing it with new features that make it even more professional.

In the first part of this chapter, you learn about threads, which enable different kinds of tasks to run concurrently. We show you how to create threads, how to start and stop them, and how to destroy them. Then you implement a new version of your ATM applet that uses threads to provide customers with advertising messages from the bank.

In the second part of this chapter, you learn how to provide national language support (NLS) to your applet. By making your applet NLS enabled, people all over the world can use it in their own language and with the correct format of specific data such as date, currency, and time information.

In order for your applet handle the differences between checking and savings accounts, you must gather information on the account selected by the user at run time. In the last part of this chapter,

you modify your applet to support run-time type information (RTTI) and to enable users to compute interest if a savings account has been selected.

Multithreading Support

Let's suppose for a moment that a bank that manages different ATM applets wants to display advertising messages. The messages would be displayed on the first panel of the ATM applet and would keep customers informed of specific offers such as low interest loans or new savings interest rates.

In this section you are going to modify your applet to provide a way of displaying messages in a ticker-type window where the text slowly pans from the right to the left like the Reuter information you see on Wall Street.

The StaticTicker Bean

Before we build a panning ticker, let's build a `StaticTicker` bean that does not scroll but is going to be very useful as a starting point. Our `StaticTicker` is built from a `Panel` class and holds the *message* property feature.

To create a simple `StaticTicker` bean, follow these steps:

1. Create a new `ch13` package in the learn Java project.
2. Create in `ch13` a new class, *StaticTicker*, which inherits from *java.awt.Panel* and choose to design the class visually. Click **Finish** to create the class.
3. Open `StaticTicker` in the Class browser and switch to the BeanInfo page.
4. Add a *read-write* property feature with *message* as the property name and *java.lang.String* as the property type. Because the bean does not need to send an event when this property is changed, you can uncheck the **bound** checkbox. Click **Finish** to create the property.
5. To display the message in the panel, add a *paint* method feature with *void* as the return type and *1* as the parameter count. Then click **Next>** to specify the argument name and type.
6. In the SmartGuide - Parameter 1 page, type in *g* as the parameter name and *java.awt.Graphics* as the parameter type. Then click **Finish** to create the method.

7. Select the paint method in the method definitions pane and modify the method in the Source pane:

```
public void paint(java.awt.Graphics g) {
    /* Perform the paint method. */
    int vOffset = size().height;
    int fontSize = getFont().getSize();
    vOffset = (vOffset - fontSize) / 2;
    g.drawString(getMessage(), 5, vOffset + fontSize);
    return;
}
```

8. Select the getMessage method and add a line to repaint the panel when the message has changed (in bold):

```
public void setMessage(String message) {
    fieldMessage = message;
    repaint();
    return;
}
```

9. Switch to the Methods page and modify each constructor to display a message prompt in the panel:

```
public StaticTicker() {
    super();
    setMessage("Your message here...");
}

public StaticTicker(java.awt.LayoutManager layout) {
    super(layout);
    setMessage("Your message here...");
}
```

10. Save your bean.

To test your bean, create a simple applet:

1. Use the Applet SmartGuide to create a *TestApplet* in the ch13 package. Choose to design the applet visually and click **Finish** to create the applet and start the Visual Composition Editor.
2. Double-click the TestApplet bean and change its layout to *BorderLayout*.
3. Add a StaticTicker bean in the center region of TestApplet. Notice that the message prompt is displayed. You can change the message and the font of the StaticTicker panel to your liking.
4. Click the **Test** button of the toolbar to save and test your applet (Figure 174).



Figure 174. StaticTicker Bean in Action

A Demanding Ticker

To make your message scroll in the panel, you need to repeatedly repaint the display string inside the ticker panel by decreasing the *x* offset. (By decreasing the *x* offset, the message scrolls from right to left). If the message goes beyond the panel boundaries, Java clips it automatically.

Let's modify the `StaticTicker` bean by adding an *xOffset* property feature and modifying the `paint` method to continuously redisplay the message with a different offset:

1. Select the `ch13.StaticTicker` bean in the Workbench.
2. From the `StaticTicker` pop-up menu, select the **Copy...** option to copy your bean in the same package.
3. A `SmartGuide` window is displayed with the `ch13` package selected and the **Rename** checkbox checked. Click **Finish** to proceed.
4. When you are prompted to enter a new name for `StaticTicker`, enter *Ticker*, and click **OK** to create the copy.
5. Select the `Ticker` bean and open the Class Browser.
6. Switch to the `BeanInfo` page.
7. Add a *read-write* property feature with *xOffset* as the property name and *int* as the property type. Because the bean does not have to send an event when this property is changed, you can uncheck the **bound** checkbox. Click **Finish** to create the property.

8. Select the paint method and modify it:

```

public void paint(java.awt.Graphics g) {
    /* Perform the paint method. */
    int yOffset = size().height;
    int fSize = getFont().getSize();
    java.awt.FontMetrics fm = getFontMetrics(getFont());
    int xMin = - fm.stringWidth(getMessage());
    yOffset = (yOffset - fSize) / 2;
    while (true) {
        try {
            Thread.sleep(10);
        }
        catch (InterruptedException e) {
            System.out.println(e);
        }
        g.drawString(getMessage(), getXOffset(),
                     yOffset + fSize);
        setXOffset(getXOffset() - 1);
        if (getXOffset() < xMin) {
            setXOffset(size().width);
        }
    }
}

```

9. Switch to the Methods page and modify each constructor to set the xOffset property:

```

public Ticker() {
    super();
    setMessage("Your message here...");
    setXOffset(size().width)
}

public Ticker(java.awt.LayoutManager layout) {
    super(layout);
    setMessage("Your message here...");
    setXOffset(size().width)
}

```

10. Save your bean.

To test your bean, modify your TestApplet:

1. Open the TestApplet bean in the Visual Composition Editor.
2. Remove the StaticTicker bean from the applet panel.
3. Add a Ticker bean in the *center* region of TestApplet. You can change the message and the font of the Ticker panel to your liking.
4. Click the **Test** button of the toolbar to save and test your applet.

What you are likely to observe by running the applet is a black trainee moving from the right to the left of your applet. In addition, if you try to resize your applet, the Ticker bean will not be resized or moved in the center of the panel. Finally, you cannot stop the applet by closing the applet viewer. To close your applet, you have to start the debugger and terminate the running thread.

When the browser or the applet viewer calls `paint` to repaint the Ticker bean, the function enters an infinite loop. The `paint` method calls the `sleep` method to pause for some amount of time (10 milliseconds in this case). The function then draws the message at the current `x` offset. Then the `x` offset is decremented and tested, in case Java has clipped the message.

Although this code looks simple, it does not work! The problem is that when the browser or the applet viewer calls the `paint` method to repaint the window, the `paint` never returns and the control is tied up in the `paint` method. For this reason the browser or the applet viewer cannot service any other events that might be occurring such as the resizing or the closing of the window. In short, your Ticker bean becomes too demanding and hogs most of the CPU time allotted to the applet.

To fix our problem, we have to separate the process of repainting the message from the rest of the program. This process is actually called a *thread* because it executes in the same memory space of the whole program.

Multithreading to the Rescue

Java supports multithreading through the `Thread` class. Thus, you can separate your program into independently running sections that execute in a thread object. By supporting multiple threads in your program, you make it more responsive to user interactions.

To create a thread in your program, you create an object of the `Thread` class. This object is used to store the state of the system when the thread does not have control. The `Thread` constructor accepts one parameter, which is an object of a class that implements the `Runnable` interface. The `Runnable` interface has only one method, `run`, which contains the section you want to execute in the separate thread. The parent thread starts the execution of the child thread by calling the `start` method from the `Thread` class. At this point, the parent thread returns from the `start` call, and the child thread begins to execute the `run` method that it implements. The parent can call the `stop` method of the `Thread` class to stop the child thread. In addition, returning from the `run` method causes the child thread to stop.

Notice that instead of creating a class that implements the `Runnable` interface, it is possible to create a class that inherits from the `Thread` class and override the `run` method. However, this second approach cannot be applied when you need to create a class that already inherits from another class such as `Applet` or `Frame`.

Let's modify the `Ticker` bean to execute the infinite loop of the `paint` method in a separate thread:

1. Open the Class Browser with the `ch13.Ticker`.
2. From the Methods page add a new field of name `paintThread` and type `java.lang.Thread`. The package access modifier is sufficient for this case. Click **Finish** to create the field.
3. From the Methods page add a `public` method, `run`, which does not take any argument and does not return anything. Click **Finish** to create the method.
4. Modify the `run` method to execute the infinite loop:

```
public void run() {
    /* Perform the run method. */
    while (true) {
        repaint();
        try {
            Thread.sleep(10);
        }
        catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

5. Modify the `paint` method (notice that the infinite loop has been removed):

```
public void paint(java.awt.Graphics g) {
    /* Perform the paint method. */
    int yOffset = size().height;
    int fSize = getFont().getSize();
    java.awt.FontMetrics fm = getFontMetrics(getFont());
    int xMin = - fm.stringWidth(getMessage());
    yOffset = (yOffset - fSize) / 2;
    g.drawString(getMessage(), getXOffset(), yOffset + fSize);
    setXOffset(getXOffset() - 1);
    if (getXOffset() < xMin) {
        setXOffset(size().width);
    }
}
```

6. Modify the Ticker class definition to implement the Runnable interface:

```
public class Ticker extends java.awt.Panel implements Runnable {
    String fieldMessage = "";
    int fieldXOffset = 0;
    Thread paintThread;
}
```

7. Modify the Ticker constructors to allocate the paintThread and start it (notice that Ticker handle is passed, through the *this* property, as the Thread constructor argument):

```
public Ticker() {
    super();
    setMessage("Your message here...");
    setXOffset(size().width)
    paintThread = new Thread(this);
    paintThread.start();
}

public Ticker(java.awt.LayoutManager layout) {
    super(layout);
    setMessage("Your message here...");
    setXOffset(size().width)
    paintThread = new Thread(this);
    paintThread.start();
}
```

8. Save your new Ticker bean.

When the applet viewer runs the applet, a Ticker object is created and creates and starts a separate paintThread. The paintThread executes the run method, which calls repaint. Repaint in turn calls paint, which displays the message at an x offset that decreases at each call until the message is clipped off to the left side of the panel. In this case, the x offset is restored to the far right of the panel.

When you run the TestApplet again, you should see the message scrolling from right to left in the applet panel (Figure 175). Notice also that you can now resize the window, and the Ticker panel is resized as well. You can now close the applet viewer that runs the applet.



Figure 175. Multithreaded Ticker in Action

Modifying the ATM Applet

You can now modify your ATM applet to display the new Ticker in the `AtmWelcomePanel` bean.

In order not to copy too many classes from one package to another, you do not copy the classes of `ch12` to `ch13`. Rather, you can version `ch12` to freeze the state of your applet as it is at the end of Chapter 12. Then you modify the classes to create a new version of your `AtmApplet`:

1. Select the `ch12` package and create a new version of the package by using the **Version...** option from its pop-up menu. Refer to “Version Control” on page 184 for more information about versioning your code.
2. Once you have versioned the `ch12` package and all its classes, open the `AtmApplet` bean in the Visual Composition Editor.
3. Add the `ch13.Ticker` bean in the north region of the applet panel.
4. Double-click the Ticker bean to open its property sheet.
5. Change its message to *Win Hawaii Trips with Sun Bank!*
6. Change its font to *dialog*, point size to *14*, and style to *bold*.
7. Edit the `FlowLayout` property and set *hgap* and *vgap* to *15*.
8. Change the foreground and background colors to your liking.
9. Save and test your applet (Figure 176).

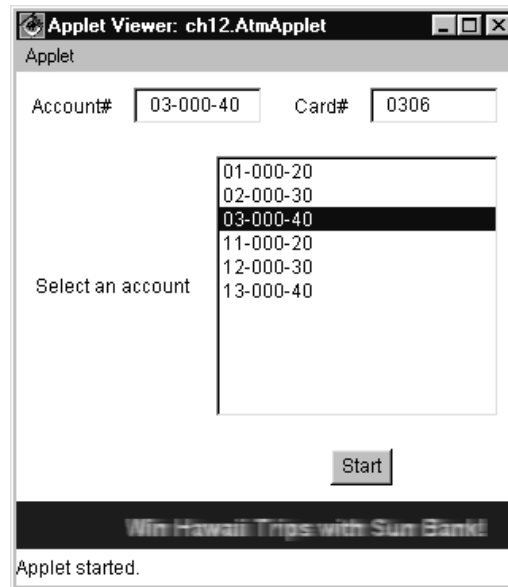


Figure 176. AtmApplet with the Ticker Bean

Resource Locking

Multithreaded programs can become very complex when you have to manage several threads at the same time, particularly if those threads access the same data. For example, one thread can update a variable that is being updated by another thread at the same time! Such *collisions* can be very difficult to track and can even appear after the program has been running for quite a while.

To prevent threads from accessing data in plain “anarchy,” Java provides a locking mechanism through the *synchronized* keyword. A method is declared synchronized when only one thread can execute it at a time. If another thread starts to enter the method, it is blocked until the previous thread has finished running the method.

In the case of our `BankAccount` class, the debit and credit methods should be synchronized to ensure that multiple credit or debit transactions cannot be carried out on the same account simultaneously. For example, the debit method should be modified as follows:

```
public synchronized void debit( Date aDate, double anAmount)
    throws InsufficientFundsException
{
    if( anAmount > getAvailableFunds()){
```

```
        throw new InsufficientFundsException(
            "Insufficient available funds: " +
            getAvailableFunds());
    }
    else {
        setBalance( getBalance() - anAmount);
        addTransaction( new Transaction( aDate, -anAmount));
    }
}
```

Mutithreaded programming is a complex subject. For more information, see the JDK documentation.

NLS Support

Today, computer applications must meet high standards. One of these standards is the adoption of a user's local customs, such as language and other conventions. This standard has become more and more critical since the Internet has exposed the world to computing and more and more people are interacting with applets through their Web browsers. Applications that support this standard are usually called NLS-enabled global or international applications.

In this section we show you what JDK 1.1 provides to help you write international applications and applets and explain how you can modify the ATM applet to take into account different country specifications.

Internationalization Framework

Writing an international application from scratch is not an easy task. Fortunately JDK 1.1 provides you with some help through an internationalization framework developed by Taligent. This framework was originally written in C++ for use in future versions of the IBM Open Class Library, but it has been ported to the Java environment. Sun has recently adopted this framework without major modifications and made it part of the official JDK 1.1.

The Java internationalization framework introduces several terms and specific classes to support them, as we explain in the sections that follow.

Locales

Java introduces the term *locale* to identify a certain geographic or political region for which spoken language and format conventions are specific. The framework defines a *java.util.Locale* class to support this term. Locale objects contain information about supported geographic or political regions.

Unlike other NLS specifications, such as that defined in POSIX/XPG4, a Locale object does not contain any locale-specific data such as number or currency formats. It serves only as an identifier for a certain geographical or political region or group.

In JDK 1.1, all locale sensitive classes use the Locale as identifier and provide locale support themselves. This approach is therefore very flexible.

To create a Locale object, you can specify a language, and optionally a country and a variant. For example, to create a Locale object for British English, you would use the following statement:

```
Locale myLocale = new Locale("en", "UK", "");
```

Obtaining All Supported Locales

Each class that performs locale-specific operations has a method that returns an array of all Locales that are supported by this class. Using this method of any of the classes, you can retrieve a list of locales (you can assume that all of the JDK 1.1 classes support all locales defined in JDK 1.1). This list of locales can then be used to list different languages associated with the locales.

As an example, let's write a simple applet that lists the different languages supported by the NumberFormat class:

1. Use the Create Class SmartGuide to create an applet of name *LocaleListApplet* in the ch13 package and open the Visual Composition Editor.
2. Drop a List bean in the middle of the panel.
3. Save and generate the code by selecting **File->Save bean** (or Ctrl+F2).
4. Switch to the Methods page and select the *init* method.
5. Insert the custom code (in bold) between the delimiters `// user code begin {1}` and `// user code end`. (This ensures that your code is not overridden when the Visual Composition Editor regenerates it):


```

public void init() {
    super.init();
    try {
        setName("LocaleListApplet");
        setLayout(null);
        setSize(426, 240);
        add(getList1(), getList1().getName());
        // user code begin {1}
        java.util.Locale[] allLocales =
            java.text.NumberFormat.getAvailableLocales();
        java.util.Locale locale =
            java.util.Locale.getDefault();
        for (int i = 0; i < allLocales.length; i++) {
            /* Check if it's a valid country */
            if (allLocales[i].getDisplayCountry(allLocales[i]).
                length() > 0) {
                /* get name of the current Locale,
                 add it to the list */
                getList1().addItem(allLocales[i].
                    getDisplayName(locale));
                /* if it's the current setting, select it */
                if (allLocales[i].getDisplayName(locale).
                    equals(locale.getDisplayName(locale))) {
                    getList1().select(i-1);
                }
            }
        }
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {2}
        // user code end
        handleException(ivjExc);
    }
}

```

- Switch back to the Visual Composition page and test the applet by clicking the **Test** button in the toolbar (Figure 177).



Figure 177. LocaleListApplet in Action

The first statement of the custom code defines an array that contains all locales supported by the `NumberFormat` class. The for loop runs through this array to add to the list box the corresponding Locale names. The `getDisplayName` method from `java.util.Locale` retrieves a human readable representation of a Locale. This method takes an optional Locale object as a parameter to determine to which locale the output should be formatted. In our case, the default locale set for the system is used as the parameter.

Refer to the JDK documentation for a list of the different locales supported.

Resource Bundles

An important point when designing international applications is to separate the program code from all locale-specific data. For example, you should separate the label of a button from the code that creates it. To achieve this design, Java provides *resource bundle* classes that store and retrieve information, using identifiers or keys.

Resource bundle classes are derived directly or indirectly from `java.text.ResourceBundle`. They are collections of resources specially designed to aggregate the resources needed for a specific geographic locale or spoken language.

A naming convention is used to identify specific resource bundle classes according to their locale, so that the resource-bundle methods knows which resource bundle classes to select on the basis of the current locale. By using inheritance among locale resources, you can minimize resource duplication across countries and achieve a graceful degradation in case the exact Locale does not have localized resources. For example, your program could use resources from generic German if there is no explicit support for Swiss German.

A locale can be set on a per-object basis, as opposed to a system wide basis, thereby making it possible to deal with more than one Locale at a time in the same program.

Because `java.text.ResourceBundle` is an abstract class, you must create your own classes that derive from it or use one of the sub-classes discussed below.

List Resource Bundles

ListResourceBundle is an abstract class which derives from ResourceBundle. It is used to store the localized data in an array of type Object. Thus localized data can be of any type.

You have to subclass ListResourceBundle with your own classes, which must override the getContents method and provide an array, where each item in the array is a pair of objects. The first element of each pair is a String key, and the second is the value associated with that key. For example, a sample ListResourceBundle class might look like this:

```
public class MyResources extends ListResourceBundle {
    public Object[] [] getContents() {
        return contents;
    }
    static final Object[] [] contents = {
        {"HelloMsg", "Hello World!"},
        {"GodbyeMsg", "Good Bye!"},
        {"BigNumber", new Integer(12345) }
    };
}
```

Assume that the above is your generic resource bundle class. Then, you would put the locale data for U.S. English there as U.S. English is the generic language of your program.

Now if you want to provide support for German also, you would create the following class:

```
public class MyResources_de extends MyResources {
    public Object[] [] getContents() {
        return contents;
    }
    static final Object[] [] contents = {
        {"HelloMsg", "Hallo Welt!"},
        {"GodbyeMsg", "Auf Wiedersehen!"},
    };
}
```

Note that the class derives from MyResources and has the name MyResources_de. This naming convention is very important because it dictates how the resources are accessed according to the current locale. In effect, when starting your program, you register with the resource bundle class loader, using MyResources. The loader then uses an automatic class lookup mechanism to find the right resource bundle class for a given locale. If the current locale is set to de, MyResources_de is used to get the resources for your program.

If you want to provide support for Swiss German, you would have to write the following class:

```
public class MyResources_de_CH extends MyResources_de {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"HelloMsg", "Grüzi Welt!"},
    };
}
```

In this case you just have to override the HelloMsg key, which is the only one to be affected. The other keys are automatically inherited from the parent class. Using this design, you build a class tree with all of your supported locales.

Property Resource Bundles

PropertyResourceBundle is an abstract subclass of ResourceBundle that manages resources for a locale by using a set of static strings from a property file. Property files have a .properties extension. They contain text lines made up with keys and their corresponding values. You use those keys in your source code in calls to ResourceBundle.getString to retrieve the associated values.

Unlike ListResourceBundle, PropertyResourceBundle can only be used to store strings, not other objects.

Loading Resource Bundles

Once you have created a class tree of resource bundles, you can load them into your Java applet or application. You use the static method getBundle from the ResourceBundle with the name of the resource bundle base class (including package information if not in the same package as the calling application) and the preferred locale. If your Locale object is called locale and you use the resource bundles classes of the previous example, your call would look like this:

```
ResourceBundle myBundle = ResourceBundle.
    getBundle("MyResources", locale);
```

In a more complex application, you would normally define many resource bundles, for example, one for each window or one for labels, one for numbers, one for pictures, one for sounds, and so on. Also, every program window might get its own bundle. In your ATM applet, you will use only one ResourceBundle.

If the current locale is set to Belgian French, the `getBundle` method first looks for a class called *MyResources_fr_BE*. If it does not find the class, it looks for generic French (*MyResources_fr*). If that search fails, the method loads the generic class *MyResources*. If the method cannot find a resource bundle or *MyResources*, it throws a `MissingResourceException`.

Retrieving Resources from Resource Bundles

Once you have successfully loaded your resource bundle class, you can access the stored objects using String keys. The `getObject` method returns an element of the static array of resources. As the resource is always returned as a `java.lang.Object`, you might have to cast it to assign the correct type to your object. For example, if you want to set an AWT label to the “HelloMsg” resource, you would have to use the following code:

```
myLabel.setText(myBundle.getString("HelloMsg"));
```

In this case, the `getString` method is equivalent to `getObject` with a casting to `String`.

Read This!



Your code must not rely on the text of certain user interface elements. In pre-JDK 1.1 programs, it was quite common to switch in the event handling using the text of the elements. This is no longer possible because the text may be different according to the active locale. Starting with JDK 1.1, you can assign names to elements to distinguish them, using the `setName` and `getName` functions that are now implemented for all AWT controls.

When you use the `ListResourceBundle` class, you can also store objects other than plain Strings. For example, you could use `myBundle` to retrieve the “BigNumber” resource into variable as follows:

```
Integer myBigNumber = (Integer)myBundle.getObject("BigNumber");
```

Formatting Data

In addition to extracting strings from resource bundles for use as identifiers (mostly in conjunction with AWT controls), you have to externalize other resources. In the sections that follow, we present an overview of the message, number, percentage, currency, date, and time formatters that you can use to help you in this matter.

Formatting Messages

One golden rule of internationalization is: *never concatenate display strings!* You may ask yourself why that could be a problem? Well, the order of parts of a sentence is different in different languages, and that can easily lead you into trouble. For example, if you write `myResourceBundle.getString("DeleteBefore") + aDate`, you are limited to modifying just the string, and not the position of the date. But, other languages might require having the date positioned at the beginning or even in the middle of the sentence.

Concatenation can be replaced by the use of the `MessageFormat` class, which enables you to position the variable information appropriately. In your message string in the resource bundle, you use placeholders for the variable. The `format` method of `MessageFormat` then substitutes the placeholders with the objects you supply. For the resource bundle classes defined in "List Resource Bundles" on page 365, you could define the following English and German strings:

```
{ "DeleteBefore", "Delete files before {0}."};
```

The German message has a different order:

```
{ "DeleteBefore", "Dateien vor dem {0} löschen." };
```

Placeholder `{0}` is for the first variable, `{1}` for the second, and so on. Using these resource strings, you can now call the `format` method to replace the placeholders with the actual values:

```
String deletePattern = myBundle.getString("DeleteBefore");
Date myDate = new Date();
String deleteMsg = MessageFormat.format(deletePattern,
                                         new Object[] {myDate});
```

Of course, you can also combine these lines into one single statement. Note that in this example, the date is not formatted according to the locale conventions. To achieve this, take a look at "Formatting Dates and Times" on page 370.

Formatting Numbers

Plain numbers can look very different for different locales. The decimal separator and hundreds separator are different for most locales and thus need to be formatted. This applies to both number output and input. JDK 1.1 supplies the `NumberFormat` class and its subclasses, `ChoiceFormat` and `DecimalFormat`, for this purpose. These classes provide formatter instances that can be used

to format relevant data. In fact, `NumberFormat` is an abstract base class from which you normally never create instances. Its factory methods automatically use one of the subclasses.

As with the `MessageFormat` class, you can use one of the factory methods of the class to get an instance using the default locale or you can supply the locale to the factory method. Formatting a certain number may look like this:

```
double myNumber = 12345.678;
NumberFormat nf = NumberFormat.getInstance(locale);
String formattedNumber = nf.format(myNumber);
```

The `DecimalFormat` subclass is used to specify a certain format for a decimal number (that is, a number of digits). This class allows you to gain even more control over the formatting numbers. You can store an object within a resource bundle to define a certain number format for each locale. Your resource bundle key and the formatting code could look like this:

```
{ "pageNumberFormat", new DecimalFormat("#,##0") }

double myNumber = 12345.678;
NumberFormat nf =
    (NumberFormat)(myBundle.getObject("PageNumberFormat"));
String formattedNumber = nf.format(myNumber);
```

Refer to the JDK documentation for a detailed explanation of the format supported by the `DecimalFormat` pattern class.

Formatting Currencies

Formatting currencies is similar to formatting plain numbers and handled by the `NumberFormat` class and its subclasses. Instead of getting the number formatting instance, you have to use another factory method to get a concrete instance that can format currencies. Look at the following simple code:

```
double myAmount = 12345.67;
NumberFormat nf = NumberFormat.getCurrencyInstance(locale);
String formattedAmount = nf.format(myAmount);
```

The `format` method returns the formatted currency, including the currency symbol (for example, 12,345,67 DM or \$12,345.67).

Formatting Percentages

Formatting percentages is similar to formatting numbers and currencies. The `NumberFormat` class provides another factory method to get a percentage formatter. The following code snippet is a sample of formatting percentages:

```
double myPercentage = 0.78;
NumberFormat nf = NumberFormat.getPercentageInstance(locale);
String formattedPercentage = nf.format(myPercentage);
```

Formatting Dates and Times

`DateFormat` and its subclasses are used to handle the formatting of date and time information. You just have to use the *getDateInstance* factory method to get the date and time formatter or *getDateInstance* to get only a date formatter. The following code shows you how to use `getDateInstance`:

```
Date myDate = new Date("21 February 1978");
DateFormat df = DateFormat.getDateInstance(DateFormat.DEFAULT,
                                           locale);
String formattedDate = df.format(myDate);
```

The first parameter in the factory method call is used to specify the format of the date or time to be used. Refer to the JDK 1.1 documentation for all possible formats.

Parsing Numbers, Currencies, Dates, and Times

You frequently have to accept user input on numbers, currencies, dates, and times. For every user input, you should provide some sort of validation. For a field that contains locale-specific data, validation can be very difficult. Fortunately, the internationalization classes of JDK provide parse functions that enable you to convert from a locale-specific format to the generic Java data type.

The following source code snippet shows how you can parse the contents of a number input field and check them for validity:

```
NumberFormat nf = NumberFormat.getInstance(locale);
try {
    double myNumber = nf.parse(getNumberTextField().getText());
}
catch (ParseException e) {
    System.out.println(e);
}
```


The same applies to currency, date, and time data. You just have to use the right instance formatter of `NumberFormat` or `DateFormat` by calling one of the factory methods.

Collation

The order of certain characters in the alphabets of different locales may be significantly different. For example, whereas in German a letter with an umlaut comes right after the regular letter (for example, `Ü` just after `U`), letters with umlauts are placed at the very end of the alphabet in Swedish. When your international program requires string comparison, you may end up with problems in correctly supporting all locales. Fortunately, JDK 1.1 has some very sophisticated techniques that help you on this issue.

With JDK 1.1 you can choose not to use comparison statements like `if (string1.equals(string2))` but rather use the abstract `Collator` class and its subclasses. The `Collator` class compares strings automatically with respect to all locale-specific issues. The class provides—like most new JDK 1.1 classes—a factory method *getInstance* that automatically returns an instance of the appropriate subclass. Take a look at the following code snippet:

```
Collator col = Collator.getInstance(locale);
if (col.equals(string1, string2)) { ..... }
if (col.compare(string1, string2) < 0 ) { ..... }
```

The first statement gets a `Collator` instance from the current locale. The second statement uses this instance to check whether two strings are equal. The third statement compares two strings and returns a negative value if the first string is less than the second (for example, `"a" < "A"`), a positive value if it is greater, and 0 if it is equal.

By using the `setStrength` method of the `Collator` class, you can specify the severity of your string comparisons. This method accepts one of the following constants of the `Collator` class ():

Table 15. String Comparison Strengths

Key Value (Constant)	Meaning
PRIMARY	When set, only primary differences are considered significant during a comparison. The assignment of strengths to language features is locale dependent. A common example is for different base letters (for example, 'a' and 'b') to be considered as primary differences.
SECONDARY	When set, only secondary differences are considered significant during a comparison. The assignment of strengths to language features is locale dependent. A common example is for different accented forms of the same letter (for example, 'a' and 'ä') to be considered as secondary differences.
TERTIARY	When set, only tertiary differences are considered significant during a comparison. The assignment of strengths to language features is locale dependent. A common example is for case differences (for example, 'a' and 'ä') to be considered as tertiary differences.
IDENTICAL	When set, all differences are considered significant during a comparison. The assignment of strengths to language features is locale dependent. A common example is for characters with equivalent Unicode spellings (for example, 'a?' and 'ä') to be considered as identical.

When you compare strings, the example above is fine because two strings are typically compared once. When you sort strings, the strings are typically compared many times, however. This method of sorting strings consumes a lot of processing time. To solve this problem, JDK 1.1 allows a string to be converted to an internal form that can be compared with a simple binary comparison. You use the *CollationKey* class, which preprocesses the string to handle all of the international issues. Take a look at this example:

```
// first build a list of sort keys
CollationKey[] keys = new CollationKey[sourceStrings.length];
for (int i = 0; i < sourceStrings.length; i++) {
    keys[i] = col.getCollationKey(sourceStrings[i]);
}
// Now sort them and add them sorted to an AWT List
sort(keys);
List list = new List();
for (int i = 0; i < sourceStrings.length; i++) {
```

```
list.addItem(keys[i].getSourceString());
}
```

Collators also support a number of advanced features such as the ability to merge in additional rules at run time or modify the rules. For example, you could make 'b' sort after 'c' if you really wanted to, or you could have '?' sort exactly as if it were spelled out as "question-mark."

Character Properties

Quite often, you have to analyze text for certain properties such as words, sentences, or paragraphs. If you need, for example, the third word in a text, you would have to extract the text located between the second and the third space characters. For sentences, you would have to check for periods, exclamation points, and question marks. This can be difficult for U.S. English and can be a daunting task for international applications. Fortunately JDK 1.1 can pull some tricks out of its sleeves to help you with these problems too.

Replacing Range Tests

If your code assumes that all characters of a given type (such as letters or digits) are those in the ASCII range, it will fail with foreign languages. Rather than test for a particular range of characters, use the Unicode character properties wherever possible. For example, instead of writing the following code:

```
for (i = 0; i < string.length(); i++) {
    char ch = string.charAt(i);
    if ((ch > 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {
        // Do something ...
    }
}
```

Use the `isLetter` static method of the `Character` class:

```
for (i = 0; i < string.length(); i++) {
    char ch = string.charAt(i);
    if (Character.isLetter(ch)) {
        // Do something ...
    }
}
```

Character Type Tests

Using the *getType* method of the *Character* class, you can easily obtain a description of a character. For example, instead of writing the following code:

```
for (i = 0; i < string.length(); i++) {
    char ch = string.charAt(i);
    if (ch == '(' || ch == '{' || ch == '[') {
        // Do something ...
    }
}
```

Use the *getType* static method of the *Character* class:

```
for (i = 0; i < string.length(); i++) {
    char ch = string.charAt(i);
    if (Character.getType(ch) == Character.START_PUNCTUATION) {
        // Do something ...
    }
}
```

Word-Break Detection

Word breaks in natural language are not just defined by spaces. For example, “spaces” from the last sentence is not followed by a space but is a word of its own. Even if you implement more sophisticated tests for ASCII text, such as checking for various punctuation, you must now deal with a wealth of possible characters in Unicode, and how they may behave differently in different countries. By using the *BreakIterator* class, you can avoid dealing with these complexities. The following example runs through a string one word at a time:

```
BreakIterator boundary = BreakIterator.getWordInstance(locale);
boundary.setText(stringToExamine);
int start = boundary.first();
for (int end = boundary.next();
     end != BreakIterator.DONE;
     start = end, end = boundary.next()) {
    System.out.println(stringToExamine.substring(start, end));
}
```

To find out whether a current index is at a word break, you can use the following code (this should be in a convenience routine in a future release of Java):

```
if (currentIndex < 0 || currentIndex > stringToExamine.length())
    return false;
if (currentIndex == 0 ||
    currentIndex == stringToExamine.length())
    return true;
```

```
int discard = boundary.following(currentIndex);
if (boundary.previous() == currentIndex)
    return true;
return false;
```

You can use different break iterators to find word boundaries, line-wrap boundaries, sentence boundaries, and character boundaries. The latter may seem mysterious: character just means Unicode character, right? However, what native users consider a single character may not be a single Unicode character, and user expectations may differ from country to country.

Converting Non-Unicode Text

All text representation in Java uses the Unicode Character Set Version 2.0 to ensure platform-independent character strings. Often, though, you have a character stream that has not been encoded by using Unicode but uses some sort of platform-specific character set. JDK provides conversion mechanisms to convert both from and to Unicode. JDK supports many character sets, but they are different on most platforms. You cannot rely on having support for the conversion of a certain character set running your Java program on a certain platform. Refer to your JDK 1.1 documentation for more information.

Creating Multilingual Applications

A multilingual application goes a bit beyond a “normal” international application. Whereas international applications support multiple locales, multilingual applications support all locales at run time. Therefore, you can change the locale of the application at run time. This change does not require special support, you just have to design your application, so that the change of the locale is instantly propagated to the whole application.

In the next section you modify the ATM applet to make part of it a multilingual applet.

Modifying the ATM Applet

To illustrate how JDK 1.1 and VisualAge for Java support the NLS framework, you are going to modify the first panel of the ATM applet to make it a multilingual panel.

To make your work easier, you import a new bean, *ImageCanvas*, in the *ch13* package. *ImageCanvas* is a canvas that allows you to display gif images.

Then you build a Panel bean, *AtmLanguagePanel*, to display the flag associated with the current locale and the system time and date information in the appropriate format. *AtmLanguagePanel* uses *ImageCanvas* to display the locale flag.

You can then modify *AtmWelcomePanel* to embed *AtmLanguagePanel*.

Once *AtmWelcomePanel* has been modified, you create a resource bundle class tree to support three locales: *fr_FR*, *de_GER*, and *us_ENG*. Then you modify the code generated by the Visual Composition Editor to update *AtmLanguagePanel* and *AtmWelcomePanel* when the locale selected by the user changes.

Importing ImageCanvas

An *ImageCanvas* bean is available in the *\Samples* directory of your CD-ROM. This bean enables you to display images in a canvas. You use this class to display the flag of the selected locale. In order for the class to work as expected, the three gif files, *us_flag.gif*, *fr_flag.gif*, and *de_flag.gif* should be placed in a directory that is accessible from your CLASSPATH. You can, for example, copy those file to your VisualAge for Java drive at *\IBM-VJava\ide\program* or in *\IBMVJava\ide\project_resources-\Learn Java*. Once you have copied the gif files, import the *ImageCanvas* bean:

1. From the Workbench menu bar, select **File->Import....** The Import SmartGuide window is displayed.
2. Type in *Learn Java* as the Project and select the Java Files radio button. Then click the **Next>** button.
3. Click **Browse...** to select *ImageCanvas.class* located in the *\Samples* directory of your CD-ROM.
4. Then click **Finish** to start importing the bean. The bean should be added to your *ch13* package.

Building AtmLanguagePanel

Now that the *ImageCanvas* bean is part of your *ch13* package, you can build *AtmLanguagePanel*:

1. Create in *ch13* an *AtmLanguagePanel* class that inherits from *java.awt.Panel*. Make sure you select the **Design the class visually** radio button to open the Visual Composition Editor.
2. Double-click *AtmLanguagePanel* to open its property sheet and change its layout to *BorderLayout*.

3. Add a Panel bean to the free-form surface, change its name to *FlagDatePanel*, and its layout to *BorderLayout*.
4. Add an ImageCanvas bean on the west region of FlagDatePanel. If the `us_flag.gif` file is copied in a directory accessible from your CLASSPATH, the American flag should be displayed.
5. Add a Panel bean to the free-form surface and change its name to *DateTimePanel* and its layout to *GridBagLayout*.
6. Add four Label beans in the DateTimePanel. Change their names to *DateLabel*, *TimeLabel*, *DateText*, *TimeText*, and their test property to *Date:*, *Time:*, *a date...*, *a time...* (Figure 178).
7. Set the grid bag constraints of the four Label beans as shown in Table 16.
8. Drag and drop DateTimePanel in the *east* region of FlagDatePanel.
9. Drag and drop FlagDatePanel in the *north* region of AtmLanguagePanel.
10. Add a Panel bean to the free-form surface, change its name to *LanguagePanel*, and its layout to *GridBagLayout*.
11. Add a Label bean and a Choice bean inside LanguagePanel and change their names to *SelectLanguageLabel* and *LanguageChoice*.
12. Change the text property of SelectLanguageLabel to *Select a Language*.
13. Set the grid bag constraints of SelectLanguageLabel and LanguageChoice as shown in Table 17.
14. Drag and drop LanguagePanel in the *south* region of AtmLanguagePanel (Figure 178).
15. Save and test AtmLanguagePanel.

Table 16. DateTimePanel GridBagLayout Constraints

Property	DateLabel	TimeLabel	DateText	TimeText
anchor	WEST	WEST	EAST	EAST
fill	HORIZONTAL	HORIZONTAL	HORIZONTAL	HORIZONTAL
gridHeight	1	1	1	1
gridWidth	1	1	1	1
gridX	0	0	1	1

Table 16. DateTimePanel GridBagLayout Constraints

Property	DateLabel	TimeLabel	DateText	TimeText
gridY	0	1	0	1
insets	T5 L5 R5 B5	T5 L5 R5 B5	T5 L5 R5 B5	T5 L5 R5 B5
ipadX	0	0	0	0
ipadY	0	0	0	0
weightX	0.5	0.5	4.0	4.0
weightY	0.0	0.0	0.0	0.0

Table 17. LanguagePanel GridBagLayout Constraints

Property	SelectLanguagePanel	LanguageChoice
anchor	CENTER	CENTER
fill	HORIZONTAL	HORIZONTAL
gridHeight	1	1
gridWidth	1	1
gridX	0	1
gridY	0	0
insets	T0 L0 R0 B0	T10 L10 R10 B10
ipadX	0	0
ipadY	0	0
weightX	0.5	1.0
weightY	0.0	0.0

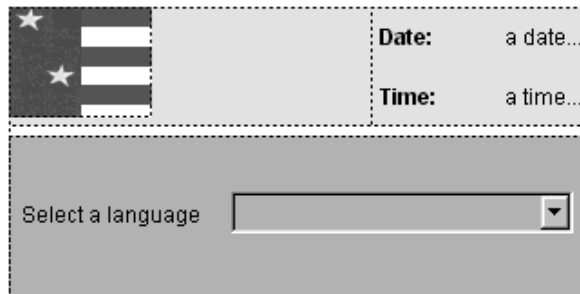


Figure 178. ATMLanguagePanel

Because you later embed `AtmLanguagePanel` in `AtmWelcomePanel`, you have to promote the `itemStateChanged(java.awt.event.ItemEvent)` event of `LanguageChoice` so that you can trigger to change the locale of `AtmWelcomePanel` when this event is sent. Do not forget to save your bean one more time.

Modifying AtmWelcomePanel

You modify `AtmWelcomePanel` directly in the `ch12` package. Before making your modifications, you can version the current state of your panel:

1. Open `AtmWelcomePanel` in the Visual Composition Editor.
2. Add an `AtmLanguagePanel` bean on top of the other controls and make it span three columns as shown in Figure 179.
3. Save and test the bean.

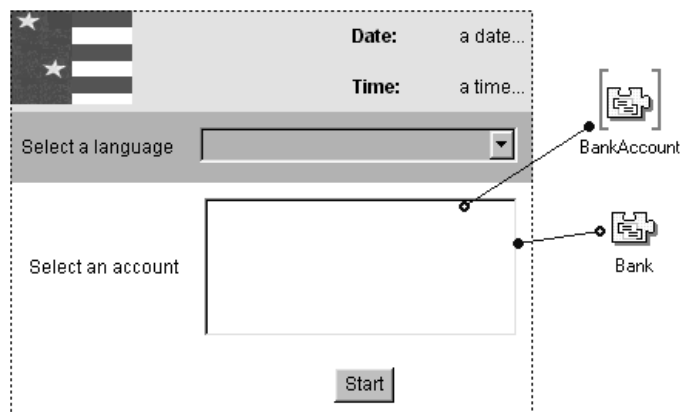


Figure 179. `AtmWelcomePanel` with `AtmLanguagePanel`

Creating the Resource Bundles

Let's now create in ch13 three resource bundle classes, *AtmResources*, *AtmResources_fr*, and *AtmResources_de*. The generic resource bundle, *AtmResources*, is used to support the default locale, *En_US*, and all the other locales that are not French or German.

Use the SmartGuide to create the following classes:

```
public class AtmResources extends java.util.ListResourceBundle {
    final static java.lang.Object[][] contents = {
        {"SelectAccountLabel", "Select an account"},
        {"StartButton", "Start"},
        {"Flag", "us_flag.gif"},
        {"SelectLanguageLabel", "Select a language"},
        {"DateLabel", "Date:"},
        {"TimeLabel", "Time:"},
    };
}

public Object[][] getContents() {
    return contents;
}

}

public class AtmResources_fr extends AtmResources {
    final static java.lang.Object[][] contents = {
        {"SelectAccountLabel", "Choisissez un compte"},
        {"StartButton", "D\u00e9buter"},
        {"Flag", "fr_flag.gif"},
        {"SelectLanguageLabel", "Choisissez une langue"},
        {"DateLabel", "Date:"},
        {"TimeLabel", "Heure:"},
    };
};

public Object[][] getContents() {
    return contents;
}

}

public class AtmResources_de extends AtmResources {
    final static java.lang.Object[][] contents = {
        {"SelectAccountLabel", "W\u00e4hlen Sie ein Konto"},
        {"StartButton", "Ok"},
        {"Flag", "de_flag.gif"},
        {"SelectLanguageLabel", "W\u00e4hlen Sie eine Sprache"},
        {"DateLabel", "Datum:"},
        {"TimeLabel", "Zeit:"},
    };
};

public Object[][] getContents() {
    return contents;
}

}
```

Notice in *AtmResources_fr* and *AtmResources_de* that some characters are defined by using their Unicode equivalent.

Updating *AtmLanguagePanel* at Run Time

To make *AtmLanguagePanel* a multilingual panel, you create three methods:

- ❑ The *changeLanguage()* method is called whenever a new language is selected by the user from *LanguageChoice*. This method translates the language selected into a supported locale and calls the *setLocale* method by passing the new locale as a parameter.
- ❑ The *setLocale(java.util.Locale)* method overrides the default *setLocale* inherited by most AWT controls since JDK 1.1. It is called first in the *initialize* method of *AtmLanguagePanel* to initialize the resources. It is also called by *changeLanguage()* each time a new language is selected. This method calls the *updateGui* method to update the GUI resources.
- ❑ The *updateGui(java.util.Locale)* method retrieves a resource bundles instance and update the GUI according to the locale passed as a parameter. Notice that the current locale can also be saved in the panel by the *setLocale* method.

To create the *changeLanguage* method:

1. Select *AtmLanguagePanel* and choose **Selected->New Method...** to start the Create Method SmartGuide.
2. Type in *void changeLanguage()* as the method name and choose *private* as the access modifier. Then click **Finish** to create the method.
3. Select the method and modify its code:

```
private void changeLanguage() {
    if (getLanguageChoice().getSelectedItem().
        equals(java.util.Locale.GERMAN.getDisplayLanguage())) {
        setLocale(java.util.Locale.GERMAN);
    }
    else
    if (getLanguageChoice().getSelectedItem().
        equals(java.util.Locale.FRANCE.getDisplayLanguage())) {
        setLocale(java.util.Locale.FRANCE);
    }
    else
    if (getLanguageChoice().getSelectedItem().
        equals(java.util.Locale.US.getDisplayLanguage())) {
        setLocale(java.util.Locale.US);
    }
    return;
}
```

To create the `setLocale` method:

1. Select `AtmLanguagePanel` and choose **Selected->New Method...** to start the Create Method SmartGuide.
2. Type in `void setLocale(java.util.Locale aLocale)` as the method name and choose `public` as the access modifier. Then click **Finish** to create the method.
3. Select the method and modify its code:

```
public void setLocale(java.util.Locale aLocale) {
    super.setLocale(aLocale);
    if (getParent() != null) {
        getParent().setLocale(aLocale);
    }
    updateGui(aLocale);
    return;
}
```

When you save the method you should get a compile error because `updateGui` has not been defined yet. Save it anyway because you are going to code `updateGui` in the next step.

Notice that the `setLocale` method changes the locale of the base class and the parent's locale. Changing the parent's locale enables you to create another multilingual panel by implementing just the `setLocale` and `updateGui` methods. No connections are needed to fire the locale change because it is taken care of by the child panel.

To create the `updateGui` method:

1. Select `AtmLanguagePanel` and choose **Selected->New Method...** to start the Create Method SmartGuide.
2. Type in `void updateGui(java.util.Locale aLocale)` as the method name and choose `private` as the access modifier. Then click **Finish** to create the method.
3. Select the method and modify its code:

```
private void updateGui(java.util.Locale aLocale) {
    java.util.ResourceBundle atmResourceBundle=null;
    // Get the proper bundle
    try {
        atmResourceBundle = java.util.ResourceBundle.
            getBundle("ch13.AtmResources", locale);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

```

// Update labels
getSelectLanguageLabel().setText(atmResourceBundle.
    getString("SelectLanguageLabel"));
getDateLabel().setText(atmResourceBundle.
    getString("DateLabel"));
getTimeLabel().setText(atmResourceBundle.
    getString("TimeLabel"));

// Update Flags
getFlag().setFlagImage(atmResourceBundle.
    getString("FlagImage"));

// Set the output format
java.text.DateFormat dFormat, tFormat;
dFormat = java.text.DateFormat.getDateInstance(0, locale);
tFormat = java.text.DateFormat.getTimeInstance(0, locale);
java.lang.String timeString =
    tFormat.format(new java.util.Date());
java.lang.String timeZone = tFormat.getTimeZone().getID();
getTimeText().setText(timeString + " " + timeZone);
java.lang.String dateString =
    dFormat.format(new java.util.Date());
getDateText().setText(dateString);
return;
}

```

At the beginning of the method, a local variable is created to hold the resource bundle object. Notice that you can create a field for this variable in your class if you want to refer to it later on. Then the resource bundle instance is retrieved, using the current locale. Once the resource bundle instance has been retrieved, it is used to get the resources for each label. At last, date and time formatters are created to format the system date and time, using the current locale.

The `setLocale` method should be called at the initialization of the panel and each time the user selects a different language.

To call `setLocale` at panel initialization, select the initialization method of `AtmLanguagePanel` and modify it (in bold):

```

private void initialize() {
    // user code begin {1}
    // user code end
    setName("AtmLanguagePanel");
    setLayout(new java.awt.BorderLayout());
    setSize(322, 160);
    this.add("Center", getLanguagePanel());
    this.add("North", getFlagDatePanel());
    initConnections();
    // user code begin {2}
    getLanguageChoice().

```

```
        add(java.util.Locale.US.getDisplayLanguage());
        getLanguageChoice().
            add(java.util.Locale.FRANCE.getDisplayLanguage());
        getLanguageChoice().
            add(java.util.Locale.GERMAN.getDisplayLanguage());
        setLocale(java.util.Locale.getDefault());
        // user code end
    }
```

Notice that you code the fill-in of the LanguageChoice and the call to setLocale with the default locale, which returns the US english locale, in between the // user code begin {2} and // user code end comment. In this way the modifications are not overridden the next time the Visual Composition Editor generates the code.

To activate the setLocale method when the user selects another language, you just add an event-to-script connection from the itemChanged event of the LanguageChoice to the changeLanguage method.

You can then save and test AtmLanguagePanel.

Updating AtmWelcomePanel at Run Time

To make AtmWelcomePanel a multilingual panel, you also create setLocale and updateGui methods. The setLocale method is called at initialization and each time the user selects a new language. Because AtmLanguagePanel is embedded in AtmWelcomePanel, the AtmWelcomePanel.setLocale method is called automatically by the AtmLanguagePanel.changeLanguage each time the user selects a different language. You do not have to add additional connections.

To create the setLocale method:

1. Select AtmWelcomePanel and choose **Selected->New Method...** to start the Create Method SmartGuide.
2. Type in *void setLocale(java.util.Locale aLocale)* as the method name and choose *public* as the access modifier. Then click **Finish** to create the method.
3. Select the method and modify its code:

```
public void setLocale(java.util.Locale aLocale) {
    super.setLocale(aLocale);
    if (getParent() != null) {
        getParent().setLocale(aLocale);
    }
    updateGui(aLocale);
    return;
}
```

When you save the method, you should get a compile error because `updateGui` has not been defined yet. Save it anyway because you are going to code `updateGui` in the next step.

To create the `updateGui` method:

1. Select `AtmLanguagePanel` and choose **Selected->New Method...** to start the Create Method SmartGuide.
2. Type in `void updateGui(java.util.Locale aLocale)` as the method name and choose `private` as the access modifier. Then click **Finish** to create the method.
3. Select the method and modify its code:

```
private void updateGui(java.util.Locale aLocale) {
    java.util.ResourceBundle atmResourceBundle=null;
    // Get the proper bundle
    try {
        atmResourceBundle = java.util.ResourceBundle.
            getBundle("ch13.AtmResources", locale);
    }
    catch (Exception e) {
        System.out.println(e);
    }

    // Update labels
    getSelectAccountLabel().setText(atmResourceBundle.
        getString("SelectAccountLabel"));

    // Update buttons
    getStartButton().setLabel(atmResourceBundle.
        getString("StartButton"));
    return;
}
```

This time the `updateGui` reflects the changes to be done for the `AtmWelcomePanel` bean only.

The `setLocale` method should also be called at the initialization of the panel. To call `setLocale` at panel initialization, select the initialization method of `AtmWelcomePanel` and modify it (in bold):

```
private void initialize() {
    // user code begin {1}
    // user code end
    java.awt.GridBagConstraints constraintsAccountList =
        new java.awt.GridBagConstraints();

    // some code ....
}
```

```

initConnections();
// user code begin {2}
getBank().initialize();
setLocale(java.util.Locale.getDefault());
// user code end
}

```

You can then save and test the multilingual `AtmWelcomePanel` (Figure 180).

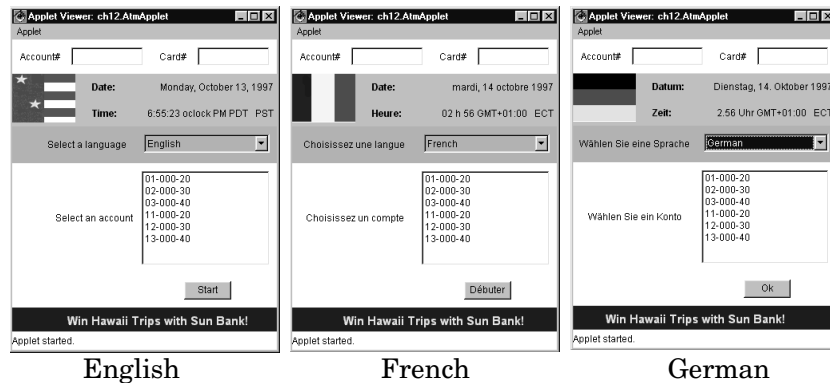


Figure 180. NLS Version of `AtmWelcomePanel` in Action

Run-Time Type Identification

RTTI lets you find the exact type of an object when you have only a handle to its base type. Java provides two forms of RTTI:

- ❑ The “traditional” RTTI assumes that you have all of the types used in your program available at compile time and run time to discover the type of an object.
- ❑ The “expanded” RTTI, also called reflection mechanism, enables you to discover class information only at run time by exploring the class features.

In this section, you use the first form of RTTI to expand the ATM applet and compute the interest earned from a selected savings account. Refer to “Introspection and `BeanInfo` Class” on page 249 for more information about the reflection mechanism.

Using Class Objects

Objects of the `Class` class represent classes and interfaces of your program. There is no public constructor for the `Class` class. `Class` objects are constructed automatically by the Java virtual machine as classes are loaded and by calls to the *defineClass* method in the class loader.

When you write a new class, a `Class` object is also created and stored, appropriately, in the `.class` file associated with your class. Then, if your program wants to make an object of that class at run time, the Java virtual machine first checks whether the `Class` object for that class is loaded. If not, it loads it by finding the `.class` file with the same name. Once the `Class` object is loaded, you can use it to query the type of the object that has just been created.

Several static methods and fields of the `Class` class can be used to query a newly created object, assuming you have a handle to it. In the ATM applet you use the `name` field to get the class name of an object for which you have only a `BankAccount` handle. Then, according to its name, you enable or disable an `Interest` button that you add to `AtmPanel` to calculate the capitalization of a savings account. For more information about the other methods and fields available from the `Class` class, refer to the JDK documentation.

Identifying the Type of Account

Remember that in `AtmWelcomePanel`, the `Bank` object initializes its list of accounts with both checking and savings accounts. Those accounts are kept in a `Vector` of type `BankAccount`. When a user selects an account, its handle is kept in a `BankAccount` variable that is used to call the `debit` and `credit` method. When you call the `credit` or `debit` method on the `BankAccount` variable, a different code of `debit` or `credit` is called, depending on whether the `BankAccount` variable refers to a `CheckingAccount` or a `SavingsAccount` object. This mechanism is called *late* or *dynamic binding* and enables polymorphism in object-oriented languages to work like magic!

Let's suppose now that you want to modify your ATM applet and enable a user to calculate the interest capitalized by a savings account he or she has selected. To provide this new function, you add one extra `Interest` button to `AtmPanel`, which triggers the *computeInterests* method you developed for the `SavingsAccount` class in Chapter 6, "Reuse in Java," on page 103.

Simple, you might say! Yes, except that you decide not to enable the Interest button of AtmPanel until you know that the BankAccount variable refers to a SavingsAccount object. This is where RTTI comes in.

To enable or disable the Interest button according to the object type referenced by the BankAccount variable, you query the class name of the object by tearing off the name property of the BankAccount variable and use the *compareTo* method of java.lang.String to compare this name with the String *ch12.CheckingAccount*:

1. Open AtmPanel in the Visual Composition Editor.
2. Add one additional Button bean to ButtonPanel.
3. Change its label to *Interest* and its name to *InterestButton*.
4. Double-click ButtonPanel and change its GridLayout setting to five rows to get the five buttons lined up in one row.
5. Select the BankAccount variable and tear off its class property. A variable of type Class is added to the free-form surface with a property-to-property connection (see **1** in Figure 181).
6. Change the Class variable to *AccountClass*. Now that you have access to the Class object, you can query its class name by tearing off its name property.
7. Tear off the name property of AccountClass. A variable of type String is added to the free-form surface with a property-to-property connection (see **2** in Figure 181).
8. Change the String variable to *ClassName*. This string must be compared to *ch12.CheckingAccount* to disable or enable InterestButton.
9. Connect the *this* property of ClassName to the *setEnabled* method of InterestButton (see **3** in Figure 181). The event-to-method connection appears dotted because a boolean parameter must be provided to execute the *setEnabled* method.
10. Connect the *b* parameter from the previous connection to the *compareTo* method of ClassName (see **4** in Figure 181). The previous connection turns plain. The parameter connection you just made appears dotted because you need to provide a string for comparison.
11. Double-click the parameter connection and click the **Set parameters...** button.
12. Type in *ch12.CheckingAccount* in the anotherString field and click **OK** to close the dialog. Click **OK** one more time to close the connection property sheet. The parameter connection should turn plain.
13. Save AtmPanel.

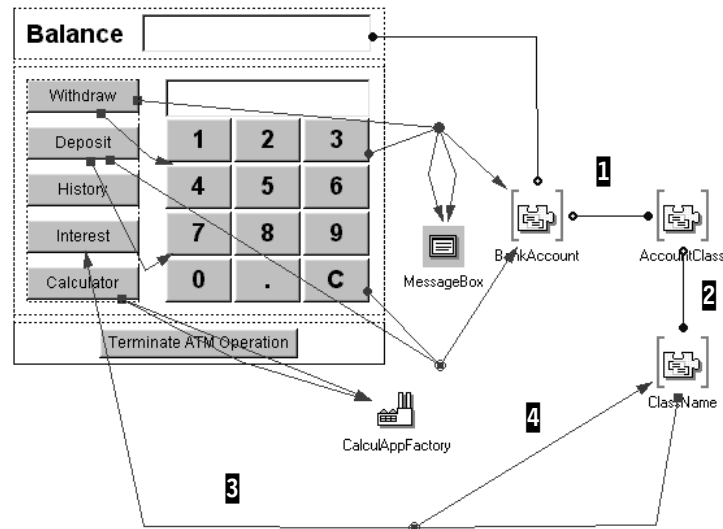


Figure 181. Identifying the Class Referenced by the BankAccountVariable

If you run `AtmApplet` and select a checking account (the first three in the list), you should have the `Interest` button disabled when you get to `AtmPanel`. Conversely, if you select a savings account (the last three in the list), you should have the `Interest` button enabled.

Accessing Methods by Downcasting

Once `InterestButton` is enabled, you have to downcast the object held by the `BankAccount` variable to access its `computeInterest` method. In effect, you cannot call this method from a `BankAccount` handle. You can call it only from a `SavingsAccount` handle or a handle on its derived classes, if there were any.

To downcast the `BankAccount` object, you copy it to another variable of type `SavingsAccount`. Then you can call the `computeInterest` from this variable:

1. Add a variable of type `ch12.SavingsAccount` to the free-form surface and change its name to *SavingsAccount*. This variable will hold the `BankAccount` object that you downcast to a `SavingsAccount` object.
2. Connect the *actionPerformed* event of `InterestButton` to the *this* property of the `SavingsAccount` variable (see 1 in Figure 182). The connection is dotted because it requires an object to initialize the `SavingsAccount` variable.

3. Connect the *this* of the `BankAccount` variable to the *value* parameter of the connection (see 2 in Figure 182). In this way, the object in the `BankAccount` variable is copied into the `SavingsAccount` variable. In other words, by making the two connections you downcast a `BankAccount` object to a `SavingsAccount` object. Now you can call the `computeInterest` method.
4. Connect the *actionPerformed* event of `InterestButton` to the `computeInterest` method of the `SavingsAccount` variable (see 3 in Figure 182).
5. Connect the *normalResult* parameter from the connection you just made to the *text* property of `BalanceTextField`, to display the interest earned (see 4 in Figure 182).
6. Save `AtmPanel` once more.

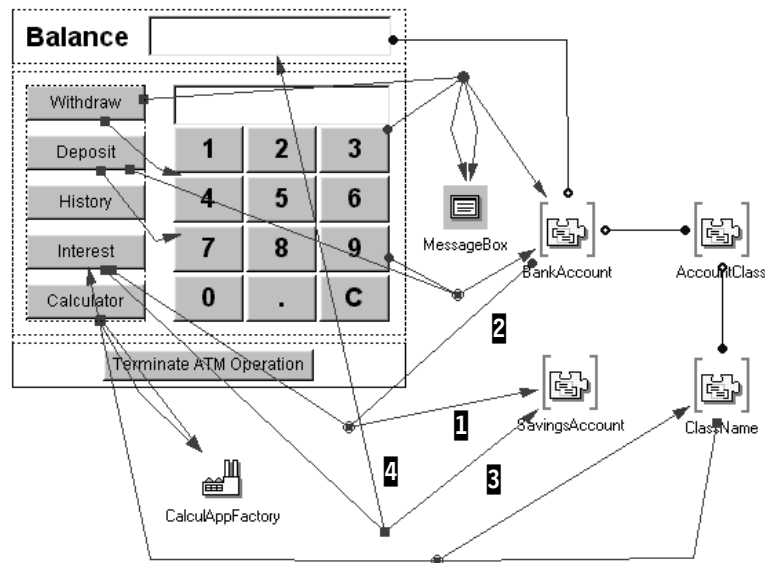


Figure 182. Downcasting a `BankAccount` Object

When you run `AtmApplet` and select a `SavingsAccount` object from the account list, the `Interest` button is enabled. If you click on it, the interest earned for the last transaction is displayed in the `Balance` text field.

Fantastic! You have finished completed (for good now!) your applet and you just need to publish it on the Internet to make it accessible to others. In the next chapter we show you how to do just that.

Summary

Using the Runnable interface, you can easily program multithreaded applets that are more responsive. JDK 1.1 provides you with an internationalization framework to help you write programs which can auto-adapt to different country specificities such as the language or the format of data. RTTI enables you to use all the power of polymorphism at run time and check the type of an object when you have a handle to its base type.

14

Publishing Your Applet

In this chapter we show you how to publish your applet on the Internet. After reading this chapter you should know how to create a simple Web page to insert your applet, how to export your class files from VisualAge for Java, and how to install the class files and other resources into your Web server's directory structure.

Exporting Your Applet

After you develop your Java applet, you may want to include it in a Web page or give the code (including JavaBeans) to colleagues. To do this, you export the applet from the workspace to the file system.

As you know from Chapter 4, “Organizing Your Code”, you can export an applet in several formats:

- ❑ **Java binary class files**, which can be executed directly by a Java virtual machine and do not allow others to access the source code of your applet
- ❑ **Java source code**, which can be compiled by a Java compiler to be executed by a Java virtual machine
- ❑ **Java archive (JAR) file**, which contains in compressed format the source code for an entire project
- ❑ **Interchange files**, which are used to exchange beans from one VisualAge for Java environment to another without losing all the development you did with the Visual Composition Editor.

To make your applet available from the Internet, you can either export its class files or export the applet and its related class files as a JAR file. When you export the applet, you must also export all of the class files of its dependent packages (other than Java class libraries).

If you do not publish the project that contains the applet or do not export the applet as a JAR file, and if the applet uses resource files, you must copy the resource files to the directory to which you export the applet files. When you run your applet from the VisualAge for Java environment, those resource files are likely to be located in the `project_resources` directory for the project that contains the applet class.

Exporting the ATM Applet As Binary Files

Let’s export the ATM applet by producing the associated .class files. In this case you select only packages `ch11`, `ch12`, and `ch13`, which contain the classes needed for the applet. In addition, you must export the `COM.ibm.ivj.javabeans` package, which contains beans such as `IList` or `IVector` that you use in your applet.

To export the ATM applet as binary files, follow these steps:

1. Select packages `ch11`, `ch12`, `ch13` and `COM.ibm.ivj.javabeans`.
2. Select **File -> Export...** from the Workbench menu bar.
3. Select the **Class Files radio** button to export only the .class files in byte-code format. Click **Next>**.
4. Type in the directory where you want to export the files and make sure the **Create package subdirectories** radio button is selected to create four directories, `ch11`, `ch12`, `ch13`, and `Com`, containing the different .class files.

5. Click **Finish** to start the export.

The directory where you export the .class files and the directory structure associated with the applet packages should be the home directory of your Web server where you usually put your Web pages. As long as the directory structure mapped from the applet packages is respected, the Java class loader will find each class with ease.

Of course, you can select a different directory. In this case you must customize your Web server to access the chosen directory. For more information about setting up your Web server, refer to “Setting Up Your Web Server” on page 400.

Once you have exported your .class files, you must move the resource files to the directory where you exported your class files. In our case you copy gif files `us_flag.gif`, `fr_flag.gif` and `de_flag.gif`, located in the `\IBM\java\ide\project_resources\Learn Java` directory in the home directory of your Web server. The resource bundle classes are part of the resources too. However, in our case those classes are already exported as .class files. If you have properties files (see “Property Resource Bundles” on page 366 for more information about properties files), you would have to copy those files to the home directory of your Web server too.

Exporting the ATM Applet As a JAR File

To export your applet as a JAR file, you export the entire Learn Java project:

1. Select the Learn Java project.
2. Select **File -> Export...** from the Workbench menu bar.
3. Select the **Jar File** radio button to export the entire project as a JAR file. Click **Next>**.
4. Type in the directory where you want to create the JAR file and the JAR file name.
5. Click **Finish** to start creating the JAR file.

Although you have created a JAR file of your applet, you still have to export the .class files of the IBM beans you are using in the applet such as `IList` or `IVector`. You can either export the `COM.ibm.ivj.javabeans` package as shown in the previous section, or you can use the `ivjbeans.jar` file provided with the Enterprise version of VisualAge for Java.

Because you are exporting your applet as a JAR file, you need not worry about the location of resources. The JAR file takes care of that by packaging the applet resources with the .class files.

Writing an HTML Page

Once you have exported your applet, you create an HTML page that embeds a reference to your applet.

Applets and the APPLET Tag

To run your applet in a browser, you must add the applet to an HTML page, using the `<APPLET>` tag. You then specify the URL of the HTML page to your browser.

Here is the simplest form of the `<APPLET>` tag:

```
<APPLET CODE=MyAppletSubClass.class WIDTH=anInt HEIGHT=anInt>
</APPLET>
```

The above tag tells the browser to load the applet whose Applet subclass, *MyAppletSubclass*, is in a class file in the same directory as the HTML document that contains the tag. The above tag also specifies the width and height in pixels of the applet.

When a browser encounters the tag, it allocates a display area of the specified width and height for the applet, loads the byte codes for the specified Applet subclass, creates an instance of the subclass, and then calls the instance's `init` and `start` methods.

Here is the complete syntax for the `APPLET` tag:

```
< APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment]
  [VSPACE = pixels] [HSPACE = pixels]
  [ARCHIVE = JARFile1, JARFile2 ...]
>
[< PARAM NAME = appletAttribute1 VALUE = value >]
[< PARAM NAME = appletAttribute2 VALUE = value >]
. . .
[alternate HTML]
</APPLET>
```

where:

- ❑ **CODEBASE = codebaseURL** is an optional attribute that specifies the base URL of the server directory that contains the applet's code. If this attribute is not specified, the document's URL is used.
- ❑ **CODE = appletFile** is a mandatory attribute that gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute.
- ❑ **ALT = alternateText** is an optional attribute that specifies any text that should be displayed if the browser understands the APPLET tag but cannot run Java applets.
- ❑ **NAME = appletInstanceName** is an optional attribute that specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.
- ❑ **WIDTH = pixels HEIGHT = pixels** are mandatory attributes that give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.
- ❑ **ALIGN = alignment** is an optional attribute that specifies the alignment of the applet. The possible values of this attribute are the same as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.
- ❑ **VSPACE = pixels HSPACE = pixels** are optional attributes that specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). They are treated the same way as the VSPACE and HSPACE attributes of the IMG tag.
- ❑ **ARCHIVE = JARFile1, JARFile2 ...** is an optional attribute that specifies one or several archive files to load. This attribute is fully discussed in the next section.
- ❑ **<PARAM NAME = appletAttribute1 VALUE = value> ...** is a tag that specifies an applet-specific attribute. Applets access their attributes with the `getParameter()` method.

A Java-enabled browser that understands the `<APPLET>` tag ignores the [Alternate HTML] part, whereas a browser that does not support Java ignores everything until [Alternate HTML]. Thus Web pages can be created that make sense for both types of browsers.

Referring to JAR Files in Your HTML Page

If you use a JAR file to keep your applet, you have two choices: You refer to the JAR file in the CLASSPATH of your machine, or you use the new ARCHIVE attribute of the <APPLET> tag to refer to the JAR file location:

```
< APPLET
...
    [ARCHIVE = JARFile1, JARFile2, ...]
...
>
```

where **ARCHIVE = JARFile1, JARFile2, ...** is an optional attribute that specifies one or more JAR files associated with the applet. The archive files should be in the same directory where the applet's class files are.

To specify that the archive file is not in the same directory as the HTML page containing the <APPLET> tag, you can use the CODEBASE attribute.

Whenever a browser has to load a file needed by an applet, it looks in the applet's JAR files first. If the browser cannot find the file in the first JAR file specified with ARCHIVE, it looks for any other specified JAR files, in the order in which they are specified. If the browser fails to find the file in a JAR file, it looks in the applet's base directory.

Any combination of JAR files and exported .class files can be used. For example, you can decide to export the ch11, ch12, and ch13 packages as .class files in separate directories and refer to the ivjbeans.jar file, using the ARCHIVE attribute in your HTML page.

If your Web server software knows how to interpret the CLASSPATH environment variable and you can modify the CLASSPATH environment variable of your Web server, you might want to avoid using the ARCHIVE attribute by adding the JAR file, specified with the ARCHIVE attribute, with its location to your CLASSPATH variable. However, this option can be very restrictive if you need to modify the CLASSPATH each time your applet has to access a new JAR file. We strongly suggest that you do not rely on the CLASSPATH environment variable of the Web server.

Sample of HTML Page for the ATM Applet

Here are two examples of HTML pages you can use to start your applet from a Java-enabled Web browser.

Exporting the ATM Applet As Binary Files

Assuming you have exported `ch11`, `ch12`, `ch13` and `COM.ibm.ivj.javabeans` packages as separate directories, you could write the following HTML page to load your applet:

```
<APPLET CODE="ch12.AtmApplet.class
        WIDTH=350 height=60
        ALT="Your browser is not Java enabled
        applet.">
    Since you cannot run the applet, here is a picture of it:
    <br>
    <IMG SRC="images/AroundTheWorld_trans.gif"
        WIDTH=316 HEIGHT=411>
</APPLET>
```

Exporting the ATM Applet As a JAR File

Assuming you have exported the entire Learn Java project as an `atm.jar` file, you are using the Enterprise version of VisualAge for Java, and that you have moved the `ivjbeans.jar` to the directory where your HTML page is located, you could write the following HTML page to load your applet:

```
<APPLET CODE="ch12.AtmApplet.class
        ARCHIVE="ivjbeans.jar, atm.jar"
        WIDTH=350 height=60
        ALT="Your browser is not Java enabled
        applet.">
    Since you cannot run the applet, here is a picture of it:
    <br>
    <IMG SRC="images/AroundTheWorld_trans.gif"
        WIDTH=316 HEIGHT=411>
</APPLET>
```

Notice that in both pages, the `ch12` package is used to reference `AtmApplet`.

Figure 183 shows the `AtmApplet` running in the Sun HotJava browser.

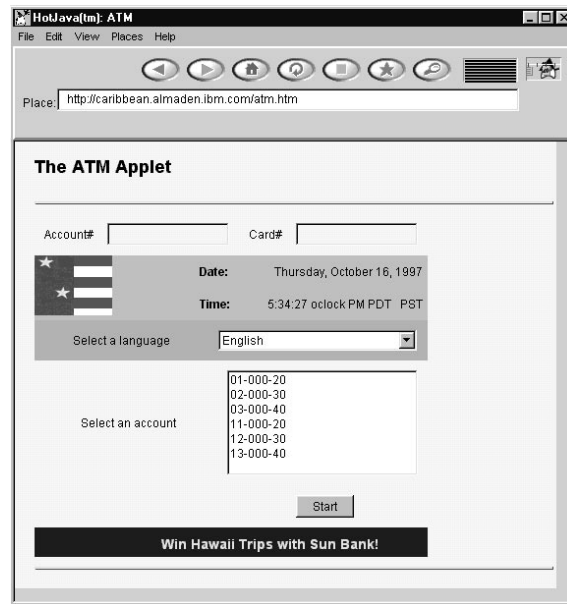


Figure 183. ATM Applet Running in the HotJava Browser

Setting Up Your Web Server

Although you may already have a Web server, such as the Microsoft Internet Information Server for Windows NT or the Microsoft Personal Web Publisher for Windows 95 installed on your machine, in this section we offer you a quick reference on installing, using, and configuring the IBM Internet Connection Server. For information about setting up the Microsoft Internet Information Server or the Microsoft Personal Web Publisher, refer to each product's user's guide.

The CD-ROM that comes with this book contains a 60 day trial version of the IBM Internet Connection Server Version 4.2 for Windows NT and a beta version of the same product for Windows 95. Other 60-day trial versions of this product are available from:

<http://www.ics.raleigh.ibm.com/icserver>

Below we describe how to set up your Web server on Windows 95 and Windows NT. For more detailed information, refer to the *IBM Internet Connection Server for Windows: User's Guide*.

Hardware and Software Requirements

Before installing the IBM Internet Connection Server, make sure you have the following hardware and software ready:

- ☐ Any personal computer or PS/2 computer with Windows 95 or Windows NT installed
- ☐ Approximately 2 MB of free disk space
- ☐ TCP/IP installed on your Windows system
- ☐ Your machine's host name, IP address, domain name, domain name server address, IP address of your router, and so on. Usually, your system administrator will have configured these for you during the installation of TCP/IP. If you are not sure of any of these, ask your system administrator.
- ☐ A LAN adapter supported by Windows. When you start your IBM Internet Connection Server, you must have your machine connected to the LAN.

Installation

The installation of the IBM Internet Connection Server is basically identical on all supported platforms. Two versions of the product are included in the CD-ROM that accompanies this book: one for Windows 95 and one for Windows NT. The installation requires six steps:

1. At an MS-DOS prompt, type:

`x:setup`

and press Enter, where x: is the drive on which you put your IBM Internet Connection Server diskette or CD-ROM. The Welcome window is displayed. Click **Next>** to proceed.

2. From the Select Components window, you need to check at least the IBM Internet Connection Server to install the product (Figure 184). Security Files can be installed to support the SSL protocol. Click **Next>** to proceed.



Figure 184. Selecting Components

From the *Choose Target Directory* window, type in the directory you want to install the IBM Internet Connection Server. The default home directory for installing the IBM Internet Connection Server is C:\WWW\, but you can change it to any other directory name you like, for example, D:\HTTPD\. Click **Next>** to continue.

3. The next two *Choose Component Directories* windows provide you with default entries for the directories of the product components that will be installed. Although we recommend that you accept the default entries, you can modify them if you want (Figure 186). Click **Next>** to proceed for each window. The installation process starts to copy files on your hard drive.

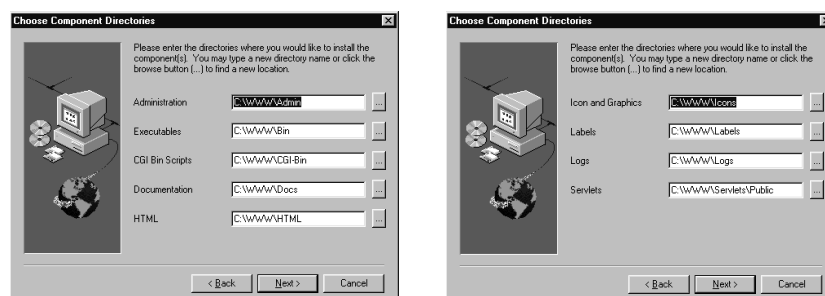


Figure 185. Selecting Directories for Your Components

4. Once the files have been copied to your disk, an Update Environment window prompts you to accept the modification of your AUTOEXEC.BAT file. Choose the first radio button to let the program update the file and click **Next>** to proceed. The

Update Environment window is displayed only if you install the Windows NT version of the IBM Internet Connection Server.

5. Once the product is installed, a message box should confirm that you must reboot your system before using the IBM Internet Connection Server (Figure 186) if you run Windows NT. If you run Windows 95, you can start the Internet Connection Server immediately.



Figure 186. Confirmation Message Box

6. Click **OK** to close the message box and the Setup program. An *IBM Internet Connection Server* service is created on your system, to be started automatically when you start your server.

Starting the Server

Using Windows NT, you can start the server by starting the IBM Internet Connection Server service from the Services icon of the Control Panel. If you use Windows 95, you can start the program from the IBM Internet Connection Server that is created in the Start menu of the taskbar.

Whether you run Windows NT or Windows 95, you can also start the IBM Internet Connection Server by entering the following command at an MS-DOS prompt:

```
WHHTTPG
```

This command starts your server with the settings you specified during installation, along with default values for other configuration settings.

Now that you have the server started, you can:

- ☐ Use the *front page* to configure the server remotely or view an HTML version of the User's Guide
- ☐ Create your own welcome page

The IBM Internet Connection Server front page gives you access to an assortment of tools and information you will find valuable as you use the server. It is an HTML document stored in a file named `Frntpage.html` on your HTML documents directory (the default is `C:\WWW\HTML`). To use the front page, open the `Frntpage.html` file from any Web browser. The front page URL is:

`http://hostname.domainname/Frntpage.html`

where `hostname.domainname` is your server's host name.

From the front page you can link to:

- ☐ *Configuration and Administration Forms*—a set of forms that you can use to easily configure your IBM Internet Connection Server to meet your particular needs
- ☐ *Sample HTML files*—a set of samples that show some of the ways you can use HTML to create your own documents
- ☐ *Documentation*—an HTML version of the *IBM Internet Connection Server for Windows User's Guide*
- ☐ *Resource list*—an HTML document containing links to many WWW sites you may find interesting and useful as you work with the IBM Internet Connection Server

Configuring the Server

The IBM Internet Connection Server comes with a default configuration that gives you a fully operational Web server. You can change any part of the default configuration to make the server meet your specific needs.

You can configure the server by using the Web server Configuration and Administration Forms or by editing the server's configuration file directly.

Using the Configuration and Administration Forms

The best and easiest way to configure your IBM Internet Connection Server is to use the Configuration and Administration Forms from the IBM Internet Connection Server front page. Using these forms, you can configure your server from the host itself or from a remote client that has access to the forms.

The URL for the main form is:

`http://hostname.domainname/admin-bin/cfgin.exe/initial`

which is a CGI program located in your server. It prompts you to enter the Administrator ID and password, which you provided when you installed your server. If you have not made any modification during installation, the default Administrator ID is **webadmin**, and the password is **webibm**. You can change the Administrator ID and password by modifying the ADMIN.PWD file, which is located in your \WINDOWS or \WINNT directory.

From the main form, you can link to each of the other input forms. Each input form lets you enter information about how you want to configure part of your server. The forms are displayed with current values in their input fields, and they provide instructions and help information to assist you in deciding what changes to make.

For each form you decide to use, make your changes to the input fields and then select the **Apply** button at the bottom of the form. The server shows you a message indicating whether your input is accepted.

If the input is accepted, the server prompts you to click on the **Restart Server** button, so that you do not need to restart your IBM Internet Connection Server manually. The changes take effect as soon as the server is restarted.

Editing the Configuration File

Another way to configure your Internet Connection Server is to edit the server's configuration file. By default, this file is named httpd.cnf. If the ETC environmental variable has been set, httpd.cnf is on that path; otherwise httpd.cnf is in the system directory, which is typically in the \WINDOWS or \WINNT directory. The configuration file is made up of statements called *directives*.

To edit the httpd.cnf file, open it using your choice of editor, update the directives, and save your changes. For a description of each of the configuration file directives, see the *Internet Connection Server for Windows NT Webmaster's Guide*. You can also view this book on line, using a browser.

Here are descriptions of some of the directives that you may find useful:

ServerRoot	Specify the home directory for the server. This directive specifies the directory where you keep the IBM Internet Connection Server executable program, HTTPD.EXE, and some other related data and programs. The default is C:\WWW\BIN, or the name you provided during installation.
-------------------	--

	You need to change this directive when you move your IBM Internet Connection Server to a different directory or a different drive.
HostName	Specify the fully qualified host name for the server. Use this directive to specify the host name of the machine on which the server is running. The default value is the same as the host name defined in your CONFIG.SYS file by SET HOSTNAME.
Port	Specify the port number you want the server to use. Use this directive to specify the port number the server should use. The default port number for HTTP is 80. Other port numbers less than 1024 are reserved for other TCP/IP applications (such as telnet, ftp, gopher) and should not be used. Port numbers 8000, 8080, 8001, and 8008 are commonly used for testing servers. When a port other than 80 is used, clients are required to include a specific port number on requests to the server, for example: <pre>http://caribbean.almaden.ibm.com:8000/</pre>
Welcome	Specify the default file displayed when the URL has a directory only. Use this directive to specify the name of a welcome file to be returned for directory requests. When the server receives a request containing a directory name, it searches the directory for files matching a name specified on a Welcome directive. You can have more than one occurrence of this directive, for example: <pre>Welcome Welcome.html Welcome Frntpage.html Welcome index.html</pre> The server might find more than one match between a file name on a Welcome directive and a file name in the requested directory. The file name from the first Welcome directive that is matched is the one that is returned.
AddType	Specify the data type of files with particular extensions.

Use this directive to bind files with a particular extension to a Multipurpose Internet Mail Extension (MIME) type or subtype. The format of the directive is:

AddType .extension type/subtype encoding [quality]

For example:

```
AddType .html text/html      8bit    1.0
AddType .au  audio/basic     binary 1.0
AddType .tif image/tiff      binary 1.0
AddType .mpeg video/mpeg     binary 1.0
AddType .mov video/quicktime binary 1.0
AddType .avi video/x-msvideo binary 1.0
AddType .gz  multipart/x-gzip binary 1.0
AddType .c   text/plain      7bit    0.5
```

Pass Specify a template for requests you want to accept and respond to with a document from your server.

Once a request matches a template on a Pass directive, the request is not compared to request templates on any subsequent directive.

In the next section, you use this directive to specify a separate directory for your Java applet data files.

Configuring the Server for Your ATM Applet

If you want people on the Internet to connect to your ATM applet, you can just export all your class files in the C:\WWW\HTML directory and copy your ATM HTML page in the same directory.

If you want to separate your ATM applet files in C:\ATM directory, for example, you can export all of the .class files and copy the ATM HTML page in this directory.

To direct the Internet Connection Server to the location of the ATM applet files, such as resources, HTML, and Java classes, you have to add a line to the server's configuration file:

1. Search in the httpd.cnf file for the Pass directive, and before the last line, which usually reads something like this:

```
Pass /*      D:\WWW\HTML\*
```

2. Add this:

```
Pass /atm/*  C:\ATM\*
```

Restart the Internet Connection Server to notify the HTTP daemon of the new directories.

Assuming that your ATM HTML page is named `atm.htm`, your ATM applet can now be accessed from the following URL:

`http://hostname.domainname/atm/atm.htm`

Web Browsers and JDK 1.1

VisualAge for Java produces JDK 1.1 Java code. To run this code, your Web browser must be Java enabled and must support JDK 1.1 or higher. At the time this book was written, few browsers met those criteria.

In this section we quickly list the main browsers you can use to run JDK 1.1-compatible Java code.

Sun HotJava 1.0

HotJava is a Web browser written by Sun Microsystems using Java itself. HotJava 1.0 supports the JavaBeans component model and can execute JDK 1.1 Java code.

You can download the HotJava browser binary free for individual, non commercial use from this URL:

`http://java.sun.com/products/hotjava`

The 1.0 feature list includes:

- HTML 3.2 compliant
- Tables
- Signed applet support
- HTTP authentication
- Left/Right "floating" table and image alignment
- New Places List model (patent pending)
- Object HTML tag support
- New optionally forgiving HTML parser
- User configurable external viewers
- Asynchronous ftp
- Persistent cookies
- JAR format
- Internationalization
- User configurable security

- Improved scrolling performance
- Improved user feedback
- Client side image maps
- Display of image and applet ALT text
- Delayed image and applet loading
- Graceful handling of low-memory environments
- Increased flexibility and an API for dynamically customizing the browser window's menus, controls, and layout from downloaded
- JDK1.1 support.

By the time this book is published, a new version of HotJava should be available from the Sun Web site. Connect to the aforementioned URL to get the latest version of this Web browser.

Netscape Communicator 4.03

The new release of Netscape Communicator 4.0.3 from Netscape Communications has been provided with a patch that allows it to run JDK 1.1 Java code. The patch is available from the developer section of the Netscape Web site at:

<http://developer.netscape.com>

The JDK 1.1 features supported by the new patch are listed below:

- JavaBeans
- AWT 1.1
- New event model
- AWT imaging enhancements
- Java Database Connection (JDBC)
- Remote Method Invocation (RMI)
- Internationalization
- Object serialization
- Inner classes
- I/O enhancements
- Reflection
- JAR file support
- Printing from an applet
- Performance enhancements
- Miscellaneous changes (Byte, Short, Void, Bignum classes)

Check the Netscape Web site frequently to download the latest patch.

Microsoft Internet Explorer 4.0

A new version of the Microsoft Internet Explorer is available for free from the Microsoft Web site at:

<http://www.microsoft.com/ie>

Microsoft Internet Explorer 4.0 does not fully support JDK 1.1 (for example, the Remote Method Invocation protocol is not supported) but you still can use it to run your ATM applet.

Compatibility Issues

Although Java has been designed from the ground up to be fully portable on all platforms for which a Java virtual machine has been developed, you may likely experience some glitches when running Java code, for these reasons:

- ❑ Java is still a new technology and needs some time to mature.
- ❑ Java virtual machines are sometimes optimized by software providers to run in specific environments. Unfortunately, those optimizations are made at a cost of compatibility.
- ❑ Web browsers are just starting to incorporate the Java virtual machine that supports the JDK 1.1 API. For this reason, Web browsers might be a little flaky.

The sample code provided with this book has been fully tested on Windows 95 and Windows NT, using various Web servers such as IBM Internet Connection Server 4.2, Microsoft Personal Web Server and Microsoft Internet Information Server 3.0, with various Web browsers such as Sun HotJava 1.0, Netscape Communicator 4.03 with the JDK 1.1 patch, and Microsoft Internet Explorer 4.0.

If you experience problems running the code provided with this book make sure the code runs at least in the appletviewer provided with your JDK. Then try to identify whether or not the problem is related to your Web server setup by checking in the Java console window for any classes that could not be loaded by the class loader. Finally, export your classes in Java source format and compile them outside the VisualAge for Java environment, using the latest JDK that you can get from Sun Microsystems at:

<http://java.sun.com/products/index.html>

Summary

You can export your applet files in various formats in order to publish your applet. By using the JAR format, you can bundle all of the classes and resources that your applet requires in one file. In addition, JAR supports compression, which reduces the file size, further improving the download time.

You must write an HTML page to embed your applet. The HTML page should be located in the proper directory of your Web server to enable users to run your applet from their Java-enabled Web browser.

15

VisualAge for Java Enterprise Edition

IBM offers VisualAge for Java in three editions:

- ❑ The *Entry* edition is the scaled-down version of the Professional edition limited to 100 user classes and can be downloaded for free from:

<http://www.software.ibm.com/java>

We provide you with this version in the CD-ROM which accompanies the book.

- ❑ The *Professional* edition comes with a robust editor, debugger, and browser and a powerful Visual Composition Editor that uses IBM's award-winning VisualAge programming paradigm.
- ❑ The *Enterprise* edition is the first enterprise-aware, incremental Java application development environment designed to connect Java clients to existing server data, transactions, and applications.

Although this book is devoted to the Professional edition of VisualAge for Java, we did not want to conclude the book without giving you a quick overview of what you can find in the Enterprise edition, and how the new features provided can improve your ATM applet.

Introduction to the Enterprise Access Builder

The Enterprise Access Builder (EAB) is a toolkit for building Java applications that can access data and programs that reside outside the Java environment on remote servers or hosts. With the EAB, you can use various builder tools that create beans which you can use with the Visual Composition Editor. The EAB supports access to relational databases, to CICS through the external call interface (ECI), and to C++ code through wrappering. The EAB also includes a proxy builder that you can use to distribute your Java classes in the network and let them communicate each other through the RMI protocol.

Relational Database Access

Enterprise Access Builder for Data (referred to as Data Access Builder) is an application development tool that you can use to create Java beans that access your existing relational database tables. You do not need to modify the source code that the Data Access Builder creates; you just use the generated beans in your programs. You invoke the Data Access Builder from the VisualAge for Java's IDE as part of a project. You must launch the Data Access Builder from a named project (not a default project).

You use Data Access Builder to generate the Java source code (classes) to access data. These Data Access classes (which are JavaBeans components) can be used directly in your Java programs or they can be used with the Visual Composition Editor in the IDE to create GUI programs. Data Access Builder also generates executable code that you can run to perform the database access testing with the generated code. As a result, you can create object-oriented applications quickly and reliably.

Some of the key features of Data Access Builder are:

- ❑ **JDBC to access your database:** Data Access Builder generates JDK Version 1.1-compliant code that uses the low-level JDBC API to access your database. You can use the JDBC driver in IBM DB2 Universal Database Version 5 Open Beta, JDBC-ODBC Bridge in JDK Version 1.1, or other JDBC drivers that support JDK V1.1 with the generated code.

- ❑ **Flexibility in specifying source:** Data Access Builder generates code from database tables, database views, or SQL statements that you enter.
- ❑ **User-friendly visual editing:** Data Access Builder uses icons for attributes and allows visual editing for such actions as deleting an attribute mapping.
- ❑ **Quick and simple to use:** You can simply specify a database table name, and Data Access Builder can access the table information and generate Java source code that enables you to add, update, delete, or retrieve rows in that table.
- ❑ **Customize to your requirements:** For special requirements, you can customize your specifications in Data Access Builder before code generation. Customization can be as simple as deleting unused attributes, or as complex as adding your own methods. A common example of customization is to change an attribute's Java data type. With the Data Access Builder, not only can you do this but you can also specify that CURRENT DATE, TIME, or TIMESTAMP be used for time-related attributes when you update or add rows in the database.
- ❑ **Data manipulation operations:** Generated classes customized to your data help you perform common database tasks such as adding, retrieving, updating, and deleting data. Classes are also generated to enable you to use a cursor to fetch rows from database queries that return result sets.
- ❑ **Executable code at your finger tips:** For rapid application development, Data Access Builder generates *AccessApp* classes, which are Java GUI applications that you can execute immediately to test connections and access databases. You can view the contents of databases and roll back changes during the test. You can use the code as is or you can use it as parts for your application.
- ❑ **Visual Composition Editor for the generated JavaBeans:** Data Access Builder generates JavaBeans compliant code that the Visual Composition Editor can use in the VisualAge for Java IDE. You can quickly create customized visual applications by using the Data Access Builder's nonvisual classes and/or the *AccessApp* classes with the Visual Composition Editor.
- ❑ **Add your own methods:** You can add your own methods to generated classes by typing SQL statements; Data Access Builder generates the Java source code for you. For example, you can add a method to update one of the tables in a table join mapping. In such situations, *AccessApp* is extremely useful in helping you test these methods as you are customizing the mapping.

- ❑ **Stored procedure support:** You can use Data Access Builder to generate code that calls stored procedures. Data Access Builder enables you to connect to the database to retrieve its stored procedure's information or to define the stored procedure in an offline mode. You can use AccessApp to immediately test stored procedure calls after code generation.
- ❑ **Generate code for table joins:** You can specify table joins using SQL statements, and Data Access Builder generates Java classes to let you process the returned result set.
- ❑ **Connection and transaction services:** Separate services are provided for connection and disconnection from your databases. In addition, commit and rollback methods are generated to handle transaction services.
- ❑ **Background thread support:** Generated methods, including methods you have defined, can run on a background thread by enabling asynchronous mode on the objects. You can improve the performance of your applications or applets by running methods asynchronously.
- ❑ **Easy-to-use samples:** A set of small Java applications is shipped with the product to enable you to quickly learn how to create mappings and use the generated code to access databases.

When Data Access Builder generates source code to access your database, it first creates a mapping from the schema, which is a database table, view, or SQL statement. The mapping contains sufficient information for Data Access Builder to generate Java classes. For example, it has definitions for all attributes matching all columns for your database table, view, or SQL statement. This mapping is then used to generate Java classes that could include add, update, delete, and retrieve methods. You can also customize the mapping, for example, by changing attribute names or by adding your own methods, before code generation.

Data Access Class Library and Generated Code

The Data Access classes enable you to connect to databases, map database tables to objects, and work with the data inside the databases by using object-oriented programming. The base classes in the Data Access Class Library provide generic methods for accessing and manipulating database information, interacting with other builders, handling messages and exceptions, and working with large objects. The generated classes extend the base classes and implement operations tailored for specific situations, as you define them in the main Data Access Builder window. In most cases, you will use the generated classes in your data access applications.

Reference Library Notation

In Table 18 and Table 19, the notation *<class>* represents the name of the Data Access class for which you are generating code. The notation *<package>* represents the name of the package in which the generated classes are located. The notation *<attribute>* represents any one of the attribute of the class, as mapped from the database table or view (corresponding to a column from the table), and *<attrType>* represents that attribute's type.

Square brackets enclosing one or more parameters in a method's signature indicate that the parameters are repeated to include all attributes of the object. For example:

```
public <class>([<attrType> an<Attribute>, ...]) throws DAException
```

Here, the parameter list will be a list of each attribute (and its type) mapped from the table to the object.

When a method name includes *<attribute>*, the generated code will include an implementation of that method for each attribute that the class has. For example, if *<class>* has two attributes, Attr1 and Attr2, and *set<attribute>* is one of the generated methods for *<class>*, two methods, *setAttr1* and *setAttr2* will be generated.

Data Access Class Library Reference

Table 18 lists the Data Access classes, which belong to the *COM.ibm.ivj.eab.data* package.

Table 18. Data Access Class Library Reference

Class Name	Description
DatastoreJDBC	Manages database connections, providing client connection to the database, disconnection from the database, and the ability to commit and roll back database transactions
PresistentObject	Provides interfaces for adding, deleting, retrieving, and updating a row from a table
PODataId	Base class for objects that uniquely identify rows in tables

Table 18. Data Access Class Library Reference

Class Name	Description
DAManager	Provides the ability to work with a collection of rows from a table and facilitates movement through a collection of rows by means of a database cursor
DAIOStream	Used for handling large object (LOB) input and output through streams
DAException	Defines exceptions thrown by the Data Access Builder and its generated code
Data Access Property Editor classes	A group of classes that let you use a GUI to edit the property values for a variety of data types. Useful when writing applications with the Visual Composition Editor

Data Access Generated Classes Reference

Table 19 lists the classes that Data Access Builder generates. Our description of the classes generated assumes that an `AtmCard` table already exists to store ATM card information.

Table 19. Classes Generated by Data Access Builder

Class Name and Example	Description
<class>Datastore (e.g., <code>AtmCardDatastore</code>)	Extends <code>DatastoreJDBC</code> . Instances of that class represent connections to the database.
<class> (e.g., <code>AtmCard</code>)	Extends <code>PersistentObject</code> . Objects represent rows from the schema. Class contains database access methods, including any user-defined methods you have defined.
<class>DataId (e.g., <code>AtmCardDataId</code>)	Extends <code>PODataId</code> . Objects represent the set of columns that uniquely identify a row. This class is generated only if the mapping specifies a data ID column or columns.

Table 19. Classes Generated by Data Access Builder

Class Name and Example	Description
<class>Manager (e.g., AtmCardManager)	Extends DAManager. Enables you to select a collection of rows from the table and work with them, or to open a database cursor and access a collection of rows one at a time. Contains any user-defined manager methods you have defined
<class>DataIdManager (e.g., AtmCardDataIdManager)	Extends DAManager. Enables you to select a collection of data IDs from the table and work with them, or to open a database cursor and access a collection of data IDs one at a time. This class is generated only if the mapping specifies a data ID column or columns.
BeanInfo classes (e.g., AtmCardBeanInfo)	Enable builders, such as the Visual Composition Editor, to use the Data Access classes by notifying the builder when an object's properties change. You will not need to use the methods in these classes directly.
Form classes (e.g., AtmCardForm)	Ready-to-use GUI parts that run the basic functions of your generated classes. You can import these into builders such as the Visual Composition Editor for use in your own applications.
AccessApp (e.g., AtmCardAccessApp)	Executable java class that runs the basic functions of your generated classes. With the simple GUIs provided by this class, you can use the generated classes to connect to datastores and manipulate data. You can use the AccessApp as an applet or application. It is useful for demonstrating your database, testing your generated classes, or accessing and manipulating database data in your final application.

CICS Access

Given the importance and widespread use of both Java and the IBM Customer Information Control System (CICS), it is not surprising to find that software developers are looking for ways to

enable their Java client programs to remotely access CICS transactions. Unfortunately, accessing a CICS transaction from Java is difficult because CICS programs are often written in COBOL and reside on MVS hosts. The obvious disparity between the Java and COBOL programming languages and the differences in internal data representation between Java workstations and MVS hosts present a real challenge to the development of any tool that can effectively enable access between Java and CICS.

However, this challenge has recently been met by the development of the CICS Access Builder. This VisualAge for Java tool makes it easy for a Java client program, run as either an applet or a stand-alone application, to remotely and seamlessly access a CICS transaction. The CICS Access Builder consists of two components: the *Create COMMAREA Bean SmartGuide* and the run-time class library.

Create COMMAREA Bean SmartGuide

The Create COMMAREA Bean SmartGuide parses the communications area of a local COBOL file that has been remotely downloaded from an MVS host. The SmartGuide imports the COBOL file and generates a COMMAREA bean and associated classes. The COMMAREA bean and associated classes contain the Java representation of the COBOL communications area, which consists of a group of records that map COBOL data to Java data, one-to-one.

When your client program is run as either an applet or an application, the COMMAREA bean is passed as a parameter to the `invokeTxn()` or `asynchInvokeTxn()` method of the `IVJCicsUOWInterface` bean. These class library methods convert the contents of the COMMAREA bean to COBOL in a form that is acceptable to the MVS host, and pass the data as part of a request to the IBM CICS Gateway for Java. The CICS Gateway for Java passes the converted data to the CICS Client, which in turn forwards the data to the CICS region on the MVS host. The CICS region invokes the CICS transaction program to process the data. Once the CICS transaction program processes the data, it sends a reply back to the client program through the CICS Client and the CICS Gateway for Java. The CICS Access Builder run-time class library then converts the COBOL data to Java.

All conversion of the data takes place on the applet or application client, so you must ensure that conversion of the communications area does not occur on the CICS server or any intermediate server. The COMMAREA data built by the Java application or applet is correct for targeting COBOL on the host.

Figure 187 shows how a CICS transaction is remotely invoked from a Java applet.

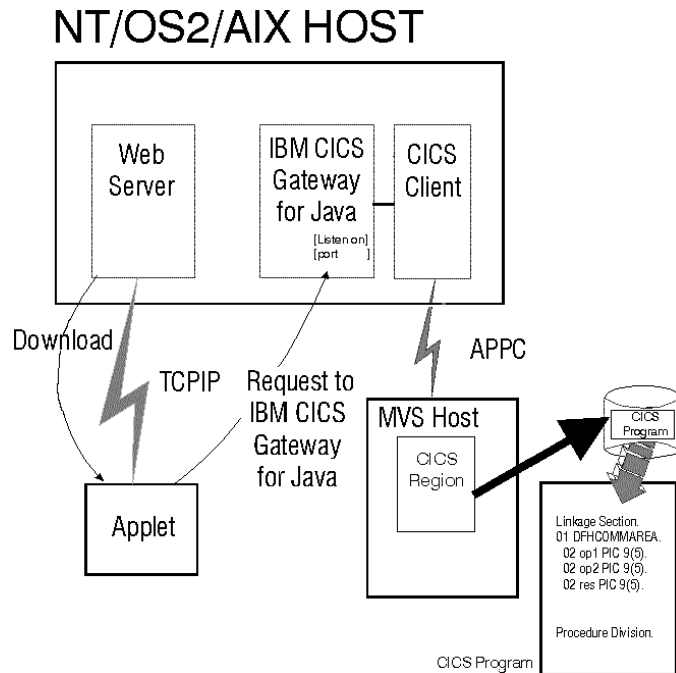


Figure 187. Invoking a CICS Transaction from a Java Applet

Run-Time Class Library

The CICS Access Builder run-time class library provides execution-time support to the COMMAREA bean and other generated classes. The class library hides the complexity of the generated code and provides the generated classes with some generic services.

Of the numerous run-time classes provided with the CICS Access Builder, the IVJCicsUOWInterface class is particularly important. This class contains `invokeTxn()` and `asynchInvokeTxn()` methods that accept the COMMAREA bean as a parameter and synchronously or asynchronously invoke a CICS transaction.

The IVJCicsUOWInterface class also contains methods that are used to manage logical units of work:

- ❑ `startUOW()` starts a unit of work.
- ❑ `endUOW()` ends a unit of work.
- ❑ `commitUOW()` commits a unit of work.
- ❑ `backoutUOW()` rolls back a unit of work.

The following scenario clarifies the relationship between the four methods of the `IVJCicsUOWInterface` class:

1. A unit of work is started by calling the `startUOW()` method.
2. One or more transactions are invoked, using the `invokeTxn()` method or the `asynchInvokeTxn()` method.
3. If a transaction does not run correctly, as evidenced by incorrect results or a return code, the unit of work can be backed out by using the `backoutUOW()` method. Once the `backoutUOW()` method has completed, any changes that were made as a result of invoking transactions are undone.
4. If all transactions run correctly and no problems are encountered, the unit of work can be committed, using the `commitUOW()` method. Once the `commitUOW()` method has completed, any changes that were made as a result of invoking transactions become permanent.

An alternative to using the `commitUOW()` method is to use the `endUOW()` method. The `endUOW()` method is used to send the commit along with the last transaction invocation request. This is accomplished by calling the `endUOW()` method on the `IVJCicsUOWInterface` class before the last `invokeTxn()` request for the unit of work. The last `invokeTxn()` method does an implicit commit.

A unit of work cannot span operations directed to different hosts, and only one request in a unit of work can be outstanding at any given time. For this reason, asynchronous requests should be used with caution.

The `IVJCicsUOWInterface` class also contains methods that are used to add and remove listeners for the following events:

- The invocation of a CICS transaction
- The start of a unit of work
- The end of a unit of work
- The commit of a unit of work
- The rollback of a unit of work
- The occurrence of an exception

These events are fired whenever the appropriate action has completed.

The `IVJCicsUOWInterface` class also contains a number of properties that you may need to set by using either the Visual Composition Editor or an editor.

Generated Classes

The CICS Access Builder generates code that enables a Java client program to remotely access a CICS transaction. The generated code includes a COMMAREA bean and associated classes.

The name generated for the COMMAREA bean is whatever name you specified in the Class Name field of the Create COMMAREA Bean SmartGuide. All other generated classes use the name generated for the COMMAREA bean as a prefix for their own names. For example, the Create COMMAREA Bean SmartGuide also generates a BeanInfo class, which uses the COMMAREA bean name as a prefix. If, for example, the name of the COMMAREA bean is myCicsTran.java, the name of the BeanInfo class is myCicsTran-BeanInfo.java.

Here are the classes generated by the CICS Access Builder:

- ❑ **classname.java:** The name of the COMMAREA bean that is the Java representation of the communications area contained in a COBOL file. The bean contains methods to set and get data in the communications area.
- ❑ **classnameBeanInfo.java:** The name of the class that provides BeanInfo information for the COMMAREA bean. It contains descriptions of methods, properties, and events in the form of beans for the COMMAREA bean.
- ❑ **classname_substructure1.java:** This class is a data member of the classname.java class that serves as the COMMAREA bean. It stores public elementary data items and calls to marshaling routines. The substructure name is the same as the name that you specified in the COMMAREA Field Name field of the Create COMMAREA Bean SmartGuide.
- ❑ **classname_substructure1_substructure2.java:** One file is generated for each substructure in the communications area to store public elementary data items and calls to marshaling routines. Above, substructure2 is the name of one of these substructures.

The following example demonstrates how classes are generated by the CICS Access Builder. Assume that in the Create COMMAREA Bean SmartGuide, you specified a class name of ADDER, a COMMAREA field name of DFHCOMMAREA, and the name of a COBOL file to import that contains the following program:

```
identification division.
program-id. ADDER.
environment division.
data division.
working-storage section.
01 tmp pic a(40).
```

```
LINKAGE SECTION.  
01 DFHCOMMAREA.  
    02 op1 PIC S99999 DISPLAY.  
    02 op2 PIC S99999 DISPLAY.  
    02 Group1.  
        03 res PIC S99999 DISPLAY.  
        03 Group2.  
            04 res2 PIC S99999 DISPLAY.  
  
procedure division.  
start-para.  
    add op1 to op2 giving res.  
    add op1 to op2 giving res2.  
    move 'ADDER transaction executed.' to tmp.  
    EXEC CICS WRITE OPERATOR TEXT(tmp) TEXTLENGTH(27)  
    ACTION(2) END-EXEC.  
    EXEC CICS RETURN  
    END-EXEC.
```

Given the information found in the above program, the CICS Access Builder generates the following classes into the project and package specified in the Create COMMAREA Bean SmartGuide:

- ADDER
- ADDERBeanInfo
- ADDER_DFHCOMMAREA
- ADDER_DFHCOMMAREA_Group1
- ADDER_DFHCOMMAREA_Group1_Group2

C++ Access

Until recently, Java software developers faced a number of problems in their quest to easily enable Java programs to access C++ services. Fortunately, these problems are now largely resolved by the C++ Access Builder. This VisualAge for Java tool makes it easy for a Java client program, run as either an applet or a stand-alone application, to access C++ services locally.

The C++ Access Builder generates code in the form of access beans and associated C++ files that enable your Java program to access existing C++ classes. It is designed to integrate C++ objects as seamlessly as possible so that programmers get the impression that the objects are actually implemented in Java. The C++ Access Builder also generates a makefile. The makefile automatically builds a library that enables C++ classes to be accessed as native methods.

The source code generated by the C++ Access Builder works in conjunction with the Java Native Interface (JNI) Specification, Release 1.1. This specification is implemented in JDK 1.1. JNI is a

native programming interface that allows Java code to interoperate with libraries written in other programming languages, such as C and C++. Although it is expected that the source code and makefile will work with multiple compilers, the source code and makefile have been tested with the IBM VisualAge for C++ compiler and the Microsoft Visual C++ compiler.

The C++ Access Builder consists of two components: a command-line interface and a run-time class library.

Command-Line Interface

The command-line interface is used to start and control the code-generation activities of the C++ Access Builder. Typically, the C++ Access Builder generates access beans and C++ wrapper files.

Run-Time Class Library

The run-time class library provides execution-time support to the access beans and associated files generated by the C++ Access Builder. The class library hides the complexity of the generated code and provides the generated code with some required generic services.

How Does the C++ Access Builder Work?

When you access another object-oriented language from Java through the procedural native code interface defined by JNI, you must address several problems:

- ☐ No JNI support for the instantiation of C++ classes
- ☐ No Java support for accessing methods and members of C++ objects
- ☐ No Java support for operator overloading
- ☐ No Java support for the passing of objects by value and reference
- ☐ No Java support for several C++ primitive types, including unsigned int and signed char

To help address these problems, the C++ Access Builder generates a Java class to represent each C++ class. It also generates a C++ wrapper that can be safely invoked through the C native method calling convention for the Java virtual machine (Figure 188). This C++ wrapper interfaces with the original C++ class.

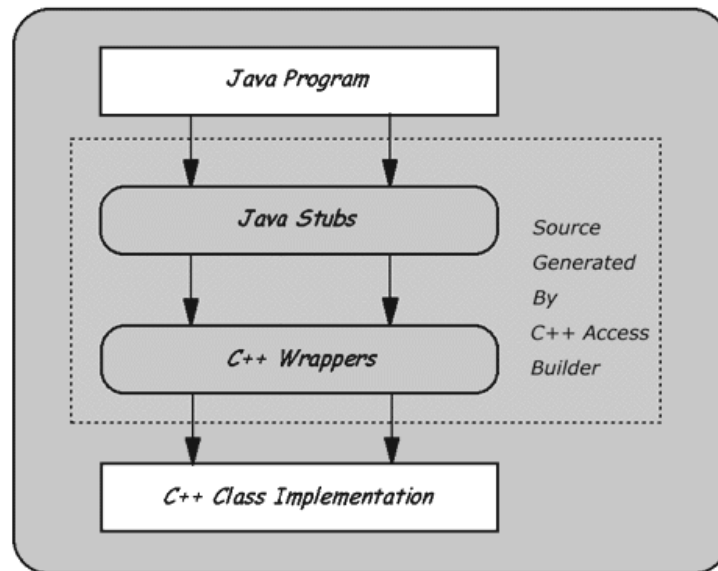


Figure 188. Java to C++ Layering

When a Java representative class is instantiated, a corresponding C++ class is instantiated, and a reference to the new C++ object is stored by the new Java object. The Java object uses this reference to invoke methods and access members of the corresponding C++ object. This bridging technique reproduces inheritance relationships between imported C++ classes on the Java side and supports polymorphic C++ methods. The C++ class methods that expect objects to be passed as parameters or to return C++ objects are accessible from Java using the Java representative objects as parameters instead.

In the case of static members, access is possible without first instantiating the object pair.

The implementation of the C++ classes to be interfaced can be provided by C++ files, shared libraries, or a combination of both. Thus the user can extend an existing C++ library by user-defined subclasses before using the C++ Access Builder to generate the Java interface.

The C++ Access Builder also maps C++ code that does not belong to a particular C++ class definition, such as overloaded nonmember operators, global functions, and global variables. This code is represented by a special Java Statics class that contains Java access methods for each C++ declaration that was not part of any C++ class.

C++ Access Builder Class Selection

Since the C++ Access Builder resolves typedef clauses and #define compiler directives, it can parse the whole hierarchy of include files for a given class definition file. Without an appropriate selection scheme indicating which classes are to be wrapped, all found C++ classes are mapped, including those found in the standard include files.

To solve this usability problem, the C++ Access Builder uses a mechanism comparable to the Java CLASSPATH. The C++ Access Builder maintains a list of directories, referred to here as J2CPP_CLASSPATH, to determine whether a found C++ class is to be mapped to Java or not.

For C++ classes that are to be mapped to Java, the following predicate must hold:

C++ class contained in class definition file contained in directory element of J2CPP_CLASSPATH

In other words, a C++ class is mapped if it is contained in a class definition file that is located in one of the member directories of J2CPP_CLASSPATH. The C++ class may be a declaration of a class, a pointer to a class, or a reference to a class.

The default class path used by the C++ Access Builder is the set of all directory prefixes of the class definition files specified on the command line, including the implicit '.' when omitted. Therefore J2CPP_CLASSPATH only needs to be set explicitly if classes are to be mapped that are defined in files other than those on the command line, such as those included using #include.

J2CPP_CLASSPATH can be modified in two ways: either by using the **-w** option on the **ivj2cpp** command, or by setting the J2CPP_CLASSPATH environment variable. The directories must be separated by a semi-colon (;).

A file named keyword.ignore is found in the help directory. It specifies any keywords that you want to ignore. If the C++ Access Builder parser cannot parse a particular word, you can direct the parser to ignore the word by adding it to the keyword.ignore file.

Generated Classes

The C++ Access Builder generates code that enables your Java program to access C++ services. This code includes access beans and C++ wrapper files. For each C++ class declared in your header file, an access bean and a C++ wrapper file is generated. The

access bean serves as a Java stub between your Java program and the C++ wrapper file. The C++ wrapper file is used to wrap the C++ class implementation.

The name of each access bean is the same as the name of the corresponding class declared in the header file. For instance, if a header file declares two classes named `myClass1` and `myClass2`, the C++ Access Builder generates two corresponding access beans named `myClass1.java` and `myClass2.java`.

Similarly, the name of each C++ wrapper file uses the name of the corresponding class declared in the header file as a name prefix. For example, if a header file declares two classes named `myClass1` and `MyClass2`, the C++ Access Builder generates two corresponding C++ wrapper files named `myClass1Wrapper.cpp` and `myClass2Wrapper.cpp`.

The C++ Access Builder also generates a makefile. The makefile automatically builds a library that enables C++ classes to be accessed as native methods.

Essentially, the only code you need to provide yourself is:

- ☐ The definitions of C++ server objects in the form of C++ header files (`.hpp`)
- ☐ The implementations in the form of C++ source files (`.cpp`)
- ☐ The Java client source code

The C++ Access Builder generates everything else. Figure 189 shows the relationship between the source code and the code generated by the C++ Access Builder.

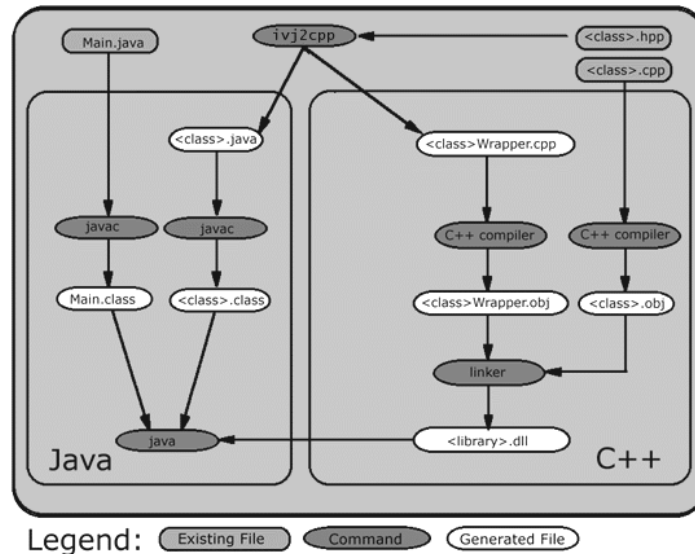


Figure 189. Relationship between the Source Code and the Generated Code

Code Generated by the C++ Access Builder

For each class declared in a specified header file, the C++ Access Builder generates the following code:

- ❑ **class.java:** Contains the Java representative bean through which C++ code is accessed
- ❑ **classWrapper.cpp:** Contains intermediate code that builds the bridge between Java and C++. Its code consists of functions called by the Java run-time system that then invokes the C++ class implementation.

For each global definition declared in a specified header file, the C++ Access Builder generates the following code:

- ❑ **Statics.java:** Contains the special Java representative class Statics. This class contains Java access methods for each C++ declaration that was not part of any C++ class. All of its methods are declared static. As a result, the Statics class does not need to be instantiated before its methods can be used.
- ❑ **StaticsWrapper.cpp:** Contains the C++ wrapper code for the Statics class.

Code Generated by the Makefile

The C++ Access Builder generates a makefile. When run, the makefile generates a system-specific library so that the generated C++ wrapper code can be called as a native method. The name of the shared library and other library files is whatever name you specify on the command parameter when you run the C++ Access Builder from the command line.

Note that Java can only use native code provided as a dynamic link library (DLL).

When the makefile is run, the following code is created:

- ❑ **sharedlib.def**: Definition file
- ❑ **sharedlib.lib**: Shared library file
- ❑ **sharedlib.dll**: DLL file
- ❑ **sharedlib.exp**: Export file on Windows 95 and NT

On OS/2, Windows 95, or Windows NT, a DLL (.dll) is needed. For this reason, the makefile generates a definition file (.def) as part of the build by using the CPPFILT utility. On Windows 95 or Windows NT, an export file (.exp) can be generated by using the VisualAge C++ ilib tool.

When you link the shared library, the JNI methods are linked by using the `javai.lib` file provided by the JDK.

Each class that contains native method definitions must have a static section in which the library is loaded. The code in the static section is executed before the constructor is called. This technique ensures that all native method definitions get assigned with the corresponding code in the library. Through their static sections, the generated Java classes will be associated with a library of this name.

The fact that the library is loaded before the object is instantiated has important implications. For instance, if you tried to introspect the generated Java bean, it would fail unless the DLL was available for loading. Therefore, the DLL should generally be built before the client is coded.

RMI Access

Until now, application developers faced several problems in developing distributed Java applications. First, it was difficult to seamlessly integrate distribution-specific code with the nondistributed,

application-specific code typically contained in a bean. Second, the object distribution protocol most commonly used to distribute a bean, Sun's RMI, does not support the distribution of bean events.

However, these and other problems have been solved by the RMI Access Builder. This VisualAge for Java tool enables you to develop distributed applications in which your Java client program, run as either an applet or client-side application, can remotely access Java server objects over RMI and invoke their methods, even if your Java code resides on a different Java virtual machine or on a different computer altogether. Furthermore, the RMI Access Builder generates distribution-specific code that seamlessly integrates with the application-specific code in your bean. It also generates code that supports the distribution of bean events.

The RMI Access Builder consists of three major components: the Create Proxy Bean SmartGuide, the run-time class library, and the Remote Object Instance Manager.

Create Proxy Bean SmartGuide

In creating a distributed application, you typically want to remotely access methods in a bean from a Java client program running on a different machine. The Create Proxy Bean SmartGuide enables you to do this by generating a proxy bean from your existing source bean. The proxy bean contains the distribution classes and interfaces required to distribute your bean over RMI. Once you have generated the proxy bean and used it to create the Java client program, you can deploy your application-specific source bean as a server bean on a remote server by instantiating your source bean through the Remote Object Instance Manager. The server bean methods are remotely invoked by your Java client program. The object instance of the server bean is called a *server object*.

The proxy bean is a client-side representative of a server bean running on a remote machine. It is used in method access and event generation activities. The proxy bean can be used in a Java client program that is implemented as either an applet or a stand-alone application, much the same as a local bean. However, although the services of the server bean appear to be carried out locally by the proxy, they are actually carried out remotely on the server system. The top-level class that implements the proxy bean is called the *client-side server proxy*.

The client-side server proxy runs on the client side of a distributed application. When the client-side server proxy is instantiated, it represents an instance of the selected server bean in that the public methods of the server bean can be accessed remotely. A client

program can use the client-side server proxy as though it is a local object, even though service requests to the client-side server proxy are actually sent to the server bean over RMI.

To support the operation of the client-side server proxy, a companion class called the *server-side server proxy* is generated to facilitate client-side server proxy communication over RMI. The server-side server proxy is generated independently of the proxy bean and must be instantiated by the Remote Object Instance Manager to function as a long-running server. The server-side server proxy must be instantiated after the RMI Registry is started, but before the client-side server proxy is put to use in client applications.

Other supporting classes and interfaces are also generated by the Create Proxy Bean SmartGuide, but the client-side server proxy and the server-side server proxy are the focal points for the client/server communication. For additional information about the proxy bean classes, and interfaces generated by the Create Proxy Bean SmartGuide, see “Generated Classes and Interfaces” on page 434.

Run-Time Class Library

The run-time class library provides execution-time support to the client-side server proxy and the server-side server proxy. The class library hides the complexity of the generated code and provides some generic services required by different client-side and server-side distribution proxies.

Remote Object Instance Manager

The Remote Object Instance Manager is a generic object instance manager for server-side server proxies generated by the RMI Access Builder. At execution time, server beans may need to provide long-running services and be instantiated in a long-running process. The Remote Object Instance Manager is a long-running process that features a console to manage the instantiation and removal of server objects (or any other Java bean objects). You can create a long-running server object instance with remote access capabilities by using the Remote Object Instance Manager to instantiate the server-side server proxy.

When the server-side server proxy is instantiated by the Remote Object Instance Manager, the server bean is then instantiated by the server-side server proxy, which creates a long-running instance of the server bean. Once the server-side server proxy and the server bean are instantiated, a client-side application or applet can access the server bean services remotely through the client-side server proxy. Different client applications on different

machines can access the same server object through the same client-side server proxy. The client-side server proxy that runs in a client machine actually represents the server object on the server system.

Special Features

The RMI Access Builder components, including the Create Proxy Bean SmartGuide, run-time class library, and Remote Object Instance Manager, together create a rich programming environment that exhibits the following features:

- ❑ If the server bean has been serialized to a serialization file, the Remote Object Instance Manager uses this file to instantiate the server object.
- ❑ Server beans are prestarted as long-running server objects by instantiating the server-side server proxy under the Remote Object Instance Manager process.
- ❑ All server bean methods that are publicly visible, including all public methods or methods defined as visible by the server bean BeanInfo class, are encapsulated as public methods in the client-side server proxy for client-side applications to invoke locally. The client-side server proxy performs Java RMI initialization and method invocation, which frees the client-side application from any RMI administrative programming tasks.
- ❑ Clients can pass both local and RMI remote objects to the client-side server proxy as parameters to the server object.
- ❑ User-defined exceptions raised by the server object can flow back to the client through the client-side server proxy as though they were actually exceptions raised by the client-side server proxy.
- ❑ All events sourced by the server object are relayed to the client-side server proxy so that the client-side server proxy can produce events on the client machine as though the server bean actually resides on the client machine. Clients can listen from the client-side server proxy for events generated by the server object. The client-side server proxy becomes the event source on the client side of the application in place of the server object. As a result, clients can be notified when bound or constrained properties are changed on the server object.
- ❑ Constrained properties are supported across the client and its associated server object.

- ❑ If a BeanInfo class is associated with the server bean, the generated client-side server proxy will also have a BeanInfo class that contains selective interface information as originally defined in the server bean BeanInfo class.

For the FeatureDescriptor part of all descriptors, the following property values from the source BeanInfo class are always reproduced in the BeanInfo class of the client-side server proxy:

- DisplayName
- ShortDescription
- Expert
- Hidden

For the EventSetDescriptor, the value of the property Unicast is reproduced in the BeanInfo class of the client-side server proxy.

For the PropertyDescriptor, the following property values from the source BeanInfo class are always reproduced in the BeanInfo class of the client-side server proxy:

- Bound
- Constrained

Generated Classes and Interfaces

The RMI Access Builder generates distribution code from an existing bean to support the distribution of your application over RMI. The generated code includes a proxy bean and individual proxy classes and interfaces. The client-side server proxy is especially important. This is the top-level class that implements the proxy bean. All other generated classes and interfaces will have the proxy prefix appended to the beginning of the class and interface names.

The name generated for the client-side server proxy is whatever name you specified in the Proxy Bean Name field of the Create Proxy Bean SmartGuide. All other generated classes use the name generated for the client-side server proxy as a prefix for their own names. For example, the Create Proxy Bean SmartGuide also generates a server-side server proxy. The server-side server proxy name uses the client-side server proxy name as a prefix but suffixes the name with an *S*. If, for example, the name of the client-side server proxy is `fooServer.java`, the name of the generated server-side server proxy is `fooServerS.java`.

Here are the classes and interfaces that the Create Proxy Bean SmartGuide generates:

- ❑ **proxyBeanName:** The name of the client-side server proxy class that implements the proxy bean. As an extended example, assume that the name of the client-side server proxy is `fooServer`. You can see how this name is used as a prefix in the sample names of the generated classes and interfaces listed below.
- ❑ **proxyBeanNameS:** The server-side server proxy, for example, `fooServerS`
- ❑ **proxyBeanNameListenerInterfaceRmiIf:** For each event class that the server object supports, the Create Proxy Bean SmartGuide creates a corresponding interface class (for example, `fooServerFooChangedListenerRmiIf`). This interface supports the transmission of events from the server-side server proxy to the client-side server proxy. The number of interfaces generated depends on the number of event listeners involved with the client-side and server-side server proxies. The application does not need to use this interface directly.
- ❑ **proxyBeanNameIf:** This class defines the server object RMI interface (for example, `fooServerIf`). It encapsulates the public methods of the server object. The class is required by the RMI services. This interface is one of the supporting classes needed by the server-side server proxy and the client-side server proxy.
- ❑ **proxyBeanNameS_Skel:** This class is the RMI skel class generated by the Create Proxy Bean SmartGuide from the server-side server proxy that is also generated by the Create Proxy Bean SmartGuide (for example, `fooServerS_Skel`). The skel class works with the RMI services in accessing remote objects. The class is one of the supporting classes needed by the server-side server proxy and the client-side server proxy.
- ❑ **proxyBeanNameS_Stub:** This class is the RMI stub class generated by the Create Proxy Bean SmartGuide from the server-side server proxy that is also generated by the Create Proxy Bean SmartGuide (for example, `fooServerS_Stub`). The stub class works with the RMI services in accessing remote objects. The class is one of the supporting classes needed by the server-side server proxy and the client-side server proxy.
- ❑ **proxyBeanName_Skel:** This class is the RMI skel class generated by the Create Proxy Bean SmartGuide from the client-side server proxy that is also generated by the Create Proxy Bean SmartGuide (for example, `fooServer_Skel`). The skel class is needed to enable the server object to call back to the client. It is generated only if the server object produces events

and has event listener registration methods. The class is one of the supporting classes needed by the server-side server proxy and the client-side server proxy.

- ❑ **proxyBeanName_Stub:** This is the RMI stub class generated by the Create Proxy Bean SmartGuide from the client-side server proxy that is also generated by the Create Proxy Bean SmartGuide (for example, `fooServer_Stub`). The stub class is needed to enable the server object to call back to the client. It is generated only if the server object produces events and has event listener registration methods. You do not have to use the stub file directly because it is used by the generated proxies, but you will have to include the file as part of the program when the application is deployed. The class is one of the supporting classes needed by the server-side server proxy and client-side server proxy.
- ❑ **proxyBeanNameBeanInfo:** This `BeanInfo` class contains information about the client-side server proxy bean interface (for example, `fooServerBeanInfo`). If there is a `fooServerBeanInfo` class, the descriptor information found in the `fooServerBeanInfo` class is transferred to this `BeanInfo` class wherever appropriate. In addition, the property descriptor and method descriptor will have descriptions on the `__serverName`, `__registryPort`, `__parent`, and `__debugTrace` properties, and related getter and setter methods.
- ❑ **proxyBeanNameEventTypeName:** This class extends the event type of an event produced by the server bean. This class is essentially a proxy class for the event class. When the server bean produces an event, the client-side server proxy creates an event of this type and invokes listeners registered against the server bean event. Client programs can access server bean event information through public methods of this event proxy.

The code fragment presented below shows how a generated client-side server proxy class named `fooServer` may be instantiated. In this example, the `fooMethod` is invoked as a local method call.

```
try {
    fooServer fs = new fooServer( component ); // create proxy
                                                // instance
    fs.fooMethod();
}
catch (IVJRException exc) {
    ... // handle exception
}

// If available, component contains the reference
// to a java.awt.Component.
// The reference is mandatory for applets.
```

The code fragment presented below uses a bean named `fooServer` to show how the classes are generated. In this example, the `fooServer` class creates `fooChangedEvent` events. The example, of course, assumes that the `fooChangedEvent` and `fooChangedListener` classes exist.

```
public class fooServer {  
    public void fooMethod() {  
        ...  
    }  
    public void addFooChangedListener(fooChangedListener l){  
        ...  
    }  
    public void removeFooChangedListener(fooChangedListener l){  
        ...  
    }  
    private void issueFooChangedEvent() {  
        ...  
    }  
}
```

Figure 190 shows how a client program interacts with a server object through the proxy bean and its support classes. The client-side server proxy and server-side server proxy use RMI services as the underlying communication mechanism between the client and the server object. Thus you can deploy the application on any Java virtual machine that supports the Java RMI programming environment.

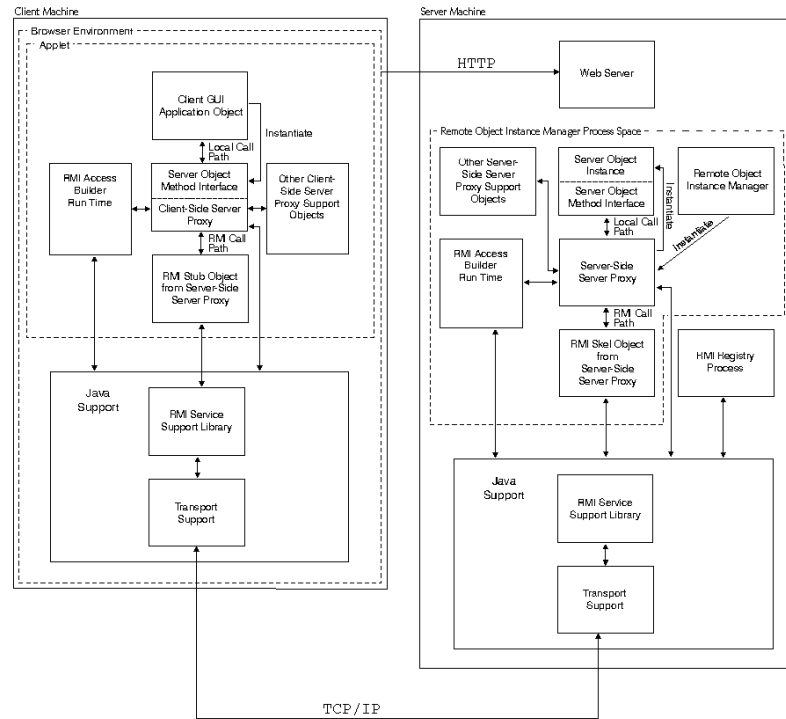


Figure 190. Client-Server Interactions through the Proxy Bean

The ATM Applet Revisited

The ATM applet cannot be run in a production environment because many critical features are missing. Although multitier architecture design is a topic beyond the scope of this book, we conclude by providing a quick overview of how the Enterprise edition of VisualAge for Java could make the ATM applet a more realistic application.

Accessing the Bank Database

To store the bank account information in a production environment, the ATM applet must have the strength of a database management system (DBMS).

Once the account and credit card information are stored in the database, the Data Access Builder can be used to create beans from the tables. Those beans can then be used in the applet to substitute the Bank class and the BankAccount variable with their Data Access Builder bean equivalent.

Accessing the Credit Card Authentication Server

You could imagine that the bank checks for credit card authentication by using a CICS transaction executed by a mainframe connected to a Credit Card Authentication Center.

This transaction could be accessed remotely from your Java applet by generating the COMMAREA bean and its associated Java classes.

Accessing Distributed Java Objects

The ATM applet should run in a light client on a computer installed in a bank agency. This computer would be connected to the central computer of the bank headquarters through TCP/IP. Java objects created at run time in the ATM applet would interact with Java objects in the central computer through RMI.

VisualAge for Java Enterprise edition would help you generate the proxies of the Bank and BankAccount to interact remotely with their surrogates in the central computer.

Home Banking

Pushing the envelop a little further, you could imagine that each bank customer would be provided with a special “black box” to use to make withdrawals and deposits. An ATM card in the box would be connected to the customer’s home computer. By connecting to their bank Web site customers would have access to the ATM applet to carry out transactions on their accounts. As you can see, home banking is not as far off as you might think!

Summary



IBM VisualAge for Java Enterprise edition is an enterprise-aware, incremental application development environment for the industry. It is designed to connect Java clients to existing server data, transactions, and applications. It enables programmers to extend server-based applications to the Internet or intranet without rewriting the applications from scratch.



VisualAge for Java Installation

This appendix explains how to install the VisualAge for Java trial version on your system and load the sample code provided with the CD-ROM in your workspace.

Installing VisualAge for Java

The CD-ROM that accompanies this book contains the VisualAge for Java Entry Edition for the Windows and OS/2 platforms. VisualAge for Java for Windows NT and Windows 95 is located in the \vajwin directory of the CD-ROM. VisualAge for Java for OS/2 is located in the \vajos2 directory of the CD-ROM.

To install VisualAge for Java on your system, follow these steps:

1. Insert the CD-ROM in your CD drive.
2. Start a command prompt and switch to the directory corresponding to your platform (Windows or OS/2).
3. Type setup and press **Enter** to start installing VisualAge for Java.
4. Once the product is installed, reboot your system.

Loading the Samples in the Workspace

The VisualAge for Java Entry Edition enables you to create a maximum of 100 classes or interfaces. For this reason, you cannot load all of the packages at one time.

We suggest that you load one package at a time and remove it from your workspace when you have completed the corresponding chapter.

To load the ch01 package in your workspace follow these steps (you would do the same with another package by substituting ch01 with another package name):

1. Create a *Learn Java* project in your workspace.
2. Select the *Learn Java* project and choose **File→Import...** from the Workbench menu bar.
3. Select the **Interchange File (Import into Repository only)** radio button and click the **Next>** button.
4. Enter X:\samples\ch01.dat in the File entry field (we assume that X: is your CD-ROM drive) and select the **Packages** radio button to list the ch01 package in the listbox.
5. Select the ch01 package and click **Finish** to load the package in your repository.
6. Select the *Learn Java* project and choose **Selected→Add Package...** from the Workbench menu bar.
7. Select the **Add package(s) from the repository** radio button and click **Browse** to access the repository.
8. Select ch01 package from the Available package names listbox.
9. Select the latest edition of the package and click the **>>** button to add the package to the Editions to Add listbox.
10. Click **OK** to start loading the package in your workspace.

Notice that in the book we create packages `ch01`, `ch02`, and so on for each chapter. Because the sample code uses the same convention, if you follow the book instructions and you have already loaded the sample code, you are likely to experience collisions with the classes loaded in your workspace. In this case, create your classes and interfaces in packages with different names. For example, instead of creating a `ch01` package, create a `chap01` package.

Notice also that because the `ch12` package uses classes that are defined in the `ch13` package, `ch13` should be loaded prior to load `ch12`.

Managing the ATM Applet Resources

Your ATM applet uses three GIF files as external resources: `de_flag.gif`, `fr_flag.gif`, and `us_flag.gif`. To run your applet from the VisualAge for Java IDE, you must copy these files located in the `\Samples` directory of the CD-ROM to the `C:\IBM\Java\ide\project_resources\Learn Java` directory.

Loading IBM Extra Beans

Starting at Chapter 5, you have to load the IBM beans (`IVector`, `IMessageBox`, and `IList`) provided with the CD-ROM.

To load the IBM beans, follow these steps:

1. Create an *IBM beans* project in your workspace.
2. Select the IBM beans project and choose **File**→**Import...** from the Workbench menu bar.
3. Select the **Entire Directory (Including resources)** radio button and click the **Next>** button.
4. Click **Browse** to select the `X:\com` directory on your CD-ROM and ensure that the **Skip .class Files (Import .java files only)** checkbox is **not** checked.
5. Click **OK** to validate the directory choice.
6. Click **Finish** to start importing the classes into your workspace.

Extra Software

The CD-ROM contains extra trial versions of software that you might want to try:

- ❑ IBM Internet Connection Server for OS/2, Windows 95, and Windows NT located in the \icsos2, \icswin95, and \icswinnt directories, respectively, of the CD-ROM. This product enables you to set up a Web server on your machine and let users download applets that you create with VisualAge for Java. Refer to “Setting Up Your Web Server on page 400” for more information about installing and setting up your IBM Internet Connection Server.
- ❑ InstallShield Java Edition located in the \ishield directory. This product enables you to create installations for all of your Java applications on all platforms—it is quick, easy and convenient. Refer to the documentation included in the \ishield directory for more information about installing this product.

B

UML Notation

This appendix shows the notation of the Unified Modeling Language (UML) which is used for the examples in this book. The following topics are covered:

- ☐ classes
- ☐ object
- ☐ inheritance
- ☐ association
- ☐ aggregation
- ☐ interaction diagram

However, this appendix is not a full reference to UML. For further information see <http://www.rational.com/>.

Class

A class is a template for creating objects. A class defines the behavior and properties that are common to all objects of that type

Graphical Representation

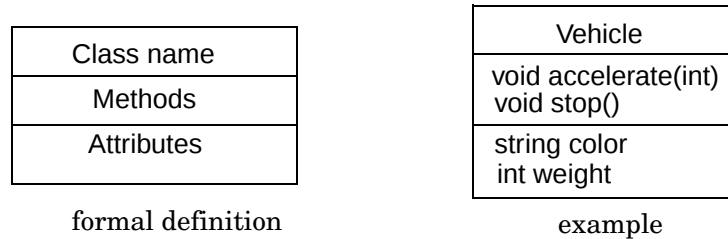


Figure 191. UML Class Notation

To show the method and attribute names is optional.

Object

An object is an instance of a class. An object shares the behavior of all objects of the same class, but each object can be in a different state

Graphical Depiction

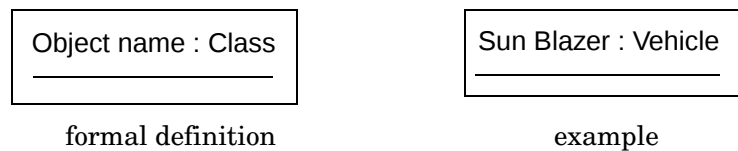


Figure 192. UML Object Notation

The object icon is similar to a class icon except that the name is underlined. To specify the class name is optional

Generalize/Inherits Relationship

Generalization helps you to share similarities among classes while preserving their differences. The term generalization refers to the relationship among classes. The term inheritance refers to the mechanism of inheriting attributes and operations using generalization.

Graphical Representation

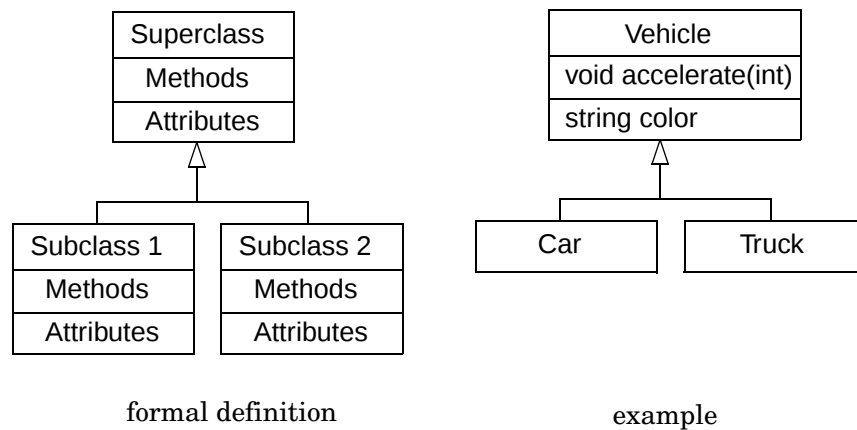


Figure 193. UML Generalization/Inheritance Relationship Notation

Association Relationship

An association relationship is a bidirectional connection between two classes. You can assign a variety of adornments to an association relationship such as association direction, roles, multiplicity, constraints and more.

Graphical Representation

You associate two classes by connecting them with a line.

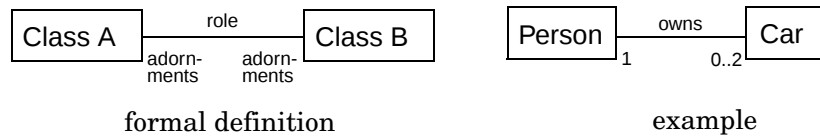


Figure 194. UML Association Relationship Notation

The role and adornments are optional. The example shows that a Person owns zero to two cars and a car belongs to one person.

Aggregation Relationship

An aggregate relationship expresses that an object (client) is physically assembled from other objects (supplier).

Graphical Representation

The aggregation relationship is represented by a line with a diamond on the side of the container.

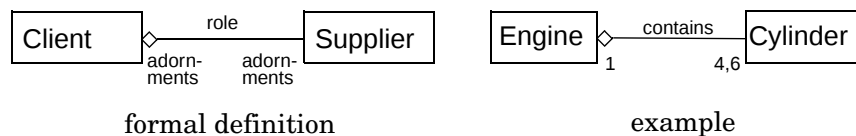


Figure 195. UML Aggregation Relationship Notation

In the example, the relationship expresses that the engine *contain* 4 or 6 cylinders.

Sequence Diagram

A sequence diagram shows the message flow between objects. The vertical dimension represents the time (proceeds normally down) and the horizontal dimension represents different objects (no specific order).

Graphical Representation

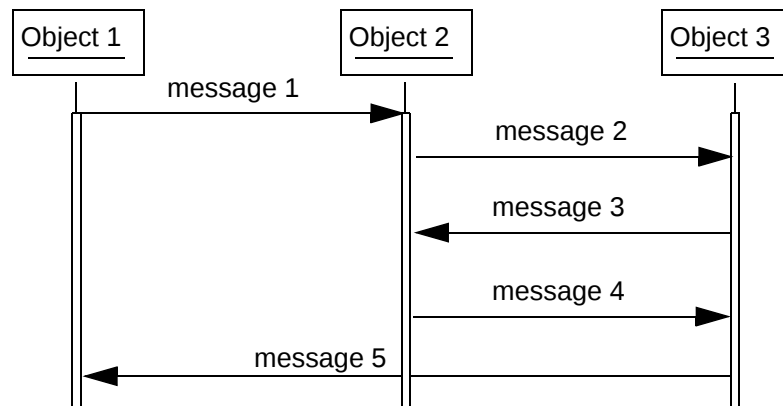


Figure 196. UML Sequence Diagram Notation



Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

International Technical Support Organization Publications

For information on ordering these ITSO publications see “How To Get ITSO Redbooks” on page 455.

- ❑ *VisualAge: Concepts and Features*, GG24-3946
- ❑ *VisualAge and Transaction Processing in a Client/Server Environment*, GG24-4487
- ❑ *Object Technology in Application Development*, GG24-4290
- ❑ *Using the Information Super Highway*, GG24-2499
- ❑ *Accessing the Internet*, GG24-2597

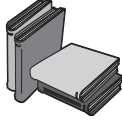
- ❑ *Creating Java Applications Using NetRexx*, SG24-2216
- ❑ *Client/Server Computing: The Design and Coding of a Business Application*, GG24-3899
- ❑ *AS/400 Application Development with VisualAge for Smalltalk*, SG24-2535
- ❑ *VisualAge: Building GUIs for Existing Applications*, GG24-4244
- ❑ *Object-Oriented Application Development with VisualAge for C++ for OS/2*, SG24-2593
- ❑ *Client Server Programming with VisualAge C++ for OS/400*, SG24-4660
- ❑ *Programming with VisualAge for C++ for Windows*, SG24-4782
- ❑ *Visual Modeling Technique, Object Technology using Visual Programming*, SG24-4227
- ❑ *VisualAge for Java—Installation Guide and Product Overview*, IBM, 1997.
- ❑ *VisualAge for Java—User's Guide*, IBM, 1997.
- ❑ *VisualAge for Java—Programming Guide*, IBM, 1997.

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RISC System/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RISC System/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection	SBOF-7250	SK2T-8042

Other Publications



These publications are also relevant as further information sources:

- ❑ *Object-Oriented Application Development with VisualAge for C++ for OS/2*, by Marc Carrel-Billiard, Peter Jakab, Isabelle Mauny, and Rainer Vetter, Prentice Hall PTR, 1996, ISBN: 0-13-242447-9
- ❑ *Programming with VisualAge for C++ for Windows*, by Marc Carrel-Billiard, Michael Friess, and Isabelle Mauny, Prentice Hall PTR, 1997, ISBN: 0-13-618208-9
- ❑ *World Wide Web Programming with VisualAge for C++ and Smalltalk*, by Andreas Bitterer and Marc Carrel-Billiard, Prentice Hall PTR, 1997, ISBN: 0-13-612466-6
- ❑ *Visual Modeling Technique*, by Daniel Tkach, Walter Fang, and Andrew So, Addison-Wesley, 1996, ISBN: 0-8053-2574-3
- ❑ *Object Technology in Application Development*, by Daniel Tkach and Richard Puttick, Benjamin/Cummings Publishing Company, ISBN: 0-8053-2572-7
- ❑ *Designing Object-Oriented Software*, by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, Prentice Hall PTR, 1994, ISBN: 0-13-629825-7
- ❑ *TCP/IP Tutorial and Technical Overview*, by Eamon Murphy, Steve Hayes, and Matthias Enders, Prentice Hall PTR, 1995, ISBN: 0-13-460858-5
- ❑ *Object-Oriented Software Engineering. A Use Case Driven Approach*, by I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, Addison-Wesley Publishing Company, 1992, ISBN: 0-201-54435-0
- ❑ *Object-Oriented Modeling and Design*, by J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Prentice Hall, 1991, ISBN: 0-13-630054-5
- ❑ *Modern Structured Analysis*, by E. Yourdon, Yourdon Press, Englewood Cliffs, New Jersey, 1989
- ❑ *Object-Oriented Analysis and Design with Applications*, by G. Booch, Benjamin/Cummings Publishing Company, 1994
- ❑ *Teach Yourself Java 1.1 in 21 Days* by L. Lemay and C. Perkins. Sams Net Publishing Company, 1997. ISBN: 1-57521142-4.
- ❑ *Java by Example* by J. Jackson and A. McClellan. SunSoft Press Java Series, 1997. ISBN: 0-13-272295-X.
- ❑ *Instant Java* by J. Pew. SunSoft Press Java Series, 1997. ISBN: 0-13-272287-X.
- ❑ *JavaBeans by Example: Cooking Beans in the Enterprise* by H. Jubin. Prentice Hall PTR, 1997. ISBN: 0-13-790338-3.

- ❑ *JDBC Developer's Resource* by A. Taylor. Prentice Hall PTR, 1997. ISBN: 0-13-842352-0.
- ❑ *Accessing CICS Business Applications from the World Wide Web* by G. De Simoni, D. Thrum, and S. Wall. Prentice Hall PTR, 1996. ISBN: 0-13-570771-4.
- ❑ *Thinking in Java* by Bruce Eckel, 1997. Available at <http://www.EckelObjects.com/Eckel>
- ❑ *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, by S. Meyers, Addison-Wesley, 1992
- ❑ *OS/2 C++ Class Library, Power GUI Programming with C Set++*, by K. Leong, W. Law, R. Love, H. Tsuji, and B. Olson, VNR Computer Library, 1993, ISBN: 0-442-01795-2
- ❑ *VisualAge for C++ for Windows—Installation Guide and Product Overview*, IBM, 1996, S33H-5030-00
- ❑ *VisualAge for C++ for Windows—User's Guide*, IBM, 1996, S33H-5031
- ❑ *VisualAge for C++ for Windows—Programming Guide*, IBM, 1996, S33H-5032
- ❑ *VisualAge for C++ for Windows—Open Class Library User's Guide*, IBM, 1996, S33H-5033
- ❑ *VisualAge for C++ for Windows—Visual Builder User's Guide*, IBM, 1996, S33H-5034
- ❑ *VisualAge for C++ for Windows—Visual Parts Reference*, IBM, 1996, S33H-5035
- ❑ *VisualAge for C++ for Windows—Building VisualAge for C++ Parts for Fun and Profit*, IBM, 1996, S33H-5036
- ❑ *VisualAge for C++ for Windows—Language Reference*, IBM, 1996, S33H-5037
- ❑ *VisualAge for C++ for Windows—C Library Reference*, IBM, 1996, S33H-5038
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume I*, IBM, 1996, S33H-5039
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume II*, IBM, 1996, S33H-5040
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume III*, IBM, 1996, S33H-5041
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume IV*, IBM, 1996, S33H-5042
- ❑ *DB2 Information and Concepts Guide*, IBM, 1995, S20H-4664
- ❑ *DB2 Administration Guide*, IBM, 1995, S20H-4580
- ❑ *DB2 Application Programming Guide*, IBM, 1995, S20H-4643

How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL <http://www.redbooks.ibm.com>.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ❑ **PUBORDER** — to order hardcopies in United States

- ❑ **GOPHER link to the Internet**
 - type GOPHER.WTSCPOK.ITSO.IBM.COM

- ❑ **Tools disks**

To get LIST3820s of redbooks, type one of the following commands:

```
TOOLS SENDTO EHON4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
```

To get lists of redbooks:

```
TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
```

To register for information on workshops, residencies, and redbooks:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
```

For a list of product area specialists in the ITSO:

```
TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
```

- ❑ **IBM Direct Publications Catalog on the World Wide Web**

<http://www.elink.ibm.link.ibm.com/pb1/pb1>

IBM employees may obtain LIST3820s of redbooks from this page.

- ❑ **REDBOOKS** category on INEWS

- ❑ **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- ☐ **Online Orders** (Do not send credit card information over the Internet)—send orders to:

	IBMMAIL	Internet
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- ☐ **Telephone orders**

United States (toll free)	1-800-879-2755	United States (toll free)
Canada (toll free)	1-800-IBM-4YOU	Canada (toll free)
Outside North America	(long distance charges apply)	Outside North America
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German	(+45) 4810-1320 - Danish
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian	(+45) 4810-1420 - Dutch
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian	(+45) 4810-1540 - English
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish	(+45) 4810-1670 - Finnish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish	(+45) 4810-1220 - French

- ☐ **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- ☐ **Fax** — send orders to:

United States (toll free)	1-800-445-9269	United States (toll free)
Canada	1-403-267-4455	Canada
Outside North America	(+45) 48 14 2207	Outside North America
	(long distance charge)	

- ☐ **1-800-IBM-4FAX (United States) or (+1) 415 855 43 29 (Outside USA)** — ask for:

Index # 4421 Abstracts of new redbooks
Index # 4422 IBM redbooks
Index # 4420 Redbooks for last six months

- ☐ **Direct Services** - send note to softwareshop@vnet.ibm.com

- ☐ **On the World Wide Web**

Redbooks Home Page <http://www.redbooks.ibm.com>
IBM Direct Publications Catalog <http://www.elink.ibm.link.ibm.com/pbl/pbl>

- ☐ **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement List-server. To initiate the service, send an E-mail note to announce@webster.ibm.link.ibm.com with the keyword `subscribe` in the body of the note (leave the subject line blank).

Special Notices

This publication is intended to help both novice and advanced Java programmers to get started using VisualAge for Java environment. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java, Database 2, or IBM Internet Connection Server. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java, Database 2, and IBM Internet Connection Server for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service.

Information in this book was developed in conjunction with use of the equipment specified and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands,

and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX®	Common User Access
CSet ++™	DB2™
CUA™	DB2/2™
DRDA™	MVS/ESA®
Presentation Manager™	IBM®
Multimedia Presentation Manager/2™	OS/2®
SOMobjects™	OS/2 Warp®
OS/400®	VisualAge™
Workplace Shell™	VisualAge for Java™
WebExplorer™	

The following terms are trademarks of other companies:

Borland® and dBase® are registered trademark of Borland International, Inc.

C-bus™ is a trademark of Collary, Inc.

HP-UX™ are trademarks of Hewlett-Packard Company.

i386™ and Pentium™ are trademarks of Intel Corporation.

Informix® is a registered trademark of Informix Software, Inc.

Ingres™ is a trademark of Ingres Corporation.

INTERSOLV™, Datadirect are trademarks of INTERSOLV Corporation.

IPX/SPX® are registered trademarks of Novell, Inc.

Sun™, Java® and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc.

HotJava™, JavaBeans™, JDK™, JDBC™, JNI™, are trademarks of Sun Microsystems, Inc.

Lotus®, Approach® are registered trademarks of Lotus Development Corporation.

Microsoft®, MS®, Access®, Excel®, Internet Explorer® are registered trademarks and ODBC™, Windows™, Windows™ NT are trademarks of Microsoft Corporation.

Motif® is a registered trademark of Open Software Foundation.

Netscape® and Netscape Navigator® are registered trademarks of Netscape Communications Corporation.

Oracle® is a registered trademark of Oracle Corporation.

PC Direct™ is a trademark of Ziff Communications Company and is used by IBM Corporation under licence.

Sinix™ is a trademark of Siemens.

Smalltalk™ is a trademark of Xerox Corporation.

Solaris® is registered trademark of Sun Microsystems.

SQLBase® is a registered trademark of Gupta Technologies.

SYBASE® is a registered trademark of Sybase, Inc.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open™ is a trademark of X/Open Company Limited.

Other trademarks are trademarks of their respective companies.

The icons used in this book are from the ClipArt Collection of the CorelDRAW! Version 5 CDROM.

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York:McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept.

See also *base class*.

access. A property of a class that determines whether a class member is accessible in an expression or declaration.

AccessApp. Generated by the Data Access Builder for each schema mapping, an executable GUI that provides access to the database using the other classes generated for the mapping.

accessor methods. Methods that an object provides to define the interface to its instance variables. The accessor method to return the value of an instance variable is often called a *get* method or *getter* method, and the accessor method to assign a value to an instance variable is called a *set* method or *setter* method.

action. See *Method*.

Compare to *attribute*, *field* and *event*.

anonymous class. In JDK 1.1, class with no name, defined in another class.

applet. A Java program designed to run within a Web browser. Contrast with application.

application. In Java programming, a self-contained, stand-alone Java program that includes *main()* method. Contrast with applet.

argument. A data element, or value, included as bean of a method call. Arguments provide additional information that the called method can use to perform the requested operation.

attribute. A specification of a property of a bean. For example, a customer bean could have a name attribute and an address attribute. An attribute can itself be a bean with its own behavior and attributes. In the Data Access Builder, the aspect of a schema mapping that represents a column in a database table.

Compare to *method* and *event*.

attribute-to-action connection. See *property-to-method connection*.

attribute-to-attribute connection. See *property-to-property connection*.

attribute-to-member function connection. See *property-to-method connection*.

B

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class.

See also *abstract class*.

bean. A definition or instance of a JavaBeans component.

See also *JavaBeans*.

BeanInfo. (1) A companion class for a bean that defines a set of methods that can be accessed to retrieve information on the bean's properties, events, and methods. (2) In the Integrated Development Environment, a page in the class browser that provides bean information.

beans palette. In the Visual Composition Editor, a two-column pane that contains prefabricated beans that you can select and manipulate to create programs. The left column contains categories of beans, while the right column contains beans for the selected category. The default set of beans generally represents JDK AWT components. You can add your own categories and beans to the beans palette.

behavior. The set of external characteristics that an object exhibits.

breakpoint. A point in a computer program where the execution may be halted.

browser. (1) In VisualAge for Java, a window that provides information on program elements. There are browsers for projects, packages, classes, methods, and interfaces. (2) An Internet-based tool that lets user browse Web sites.

C

call level interface (CLI). A callable application program interface (API) for database access, which is an alternative to an embedded SQL application program interface. In contrast to embedded SQL, CLI does not require precompiling or binding by the user, but instead provides a standard set of functions to process SQL statements and related services at run time.

caller. An object that sends a method call to another object.

Contrast with *receiver*.

Canvas. Canvases are windows with a layout algorithm that manages child windows. The Canvas class allows you to implement dialog boxes. Canvases can manage the size and position of child windows, provide moveable split bars between windows, and support the ability to scroll a window.

category. In the Visual Composition Editor, a selectable grouping of beans represented by an icon in the leftmost column. Selecting a category displays the beans belonging to that category in the next column.

See also *beans palette*.

CICS Access Builder. A VisualAge for Java Enterprise tool that generates beans to access CICS transactions through the CICS Gateway for Java and CICS Client.

CICS Client. A server program that processes CICS ECI calls, forwarding transaction requests to a CICS program running on a host.

CICS ECI. An API that provides C and C++ programs with procedural access to transactions.

CICS Gateway for Java. A server program that processes Java ECI calls and forwards CICS ECI calls to the CICS Client.

class. An aggregate that defines properties, operations, and behavior for all instances of that aggregate.

class hierarchy. The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

class library. A collection of classes.

class method. See *method*.

class path. When running a program in VisualAge for Java, a list of resource directories delimited by semi-colons.

CLASSPATH. In your deployment environment, the environment variable that specifies the directories to look for class and resource files.

CLI. See *Call Level Interface*

client area object. An intermediate window between a frame window (Frame) and its controls and other child windows.

client object. An object that requests services from other objects.

client/server. The model of interaction in distributed data processing in which a program at one location sends a request to a program at another location and awaits a response. The requesting program is called a client, and the answering program is called a server.

client-side server proxy. Generated by the RMI Access Builder, a local representative of a remote bean. This proxy provides access to the operations of the server bean, allowing a Java client to work with it as if it were the server bean.

See also *proxy bean* and *server-side server proxy*.

Class Browser. In the VisualAge for Java IDE, a tool used to browse the classes loaded in the workspace.

collection. A set of features in which each feature is an object.

COM. See *Compound Object Model*.

commit. The operation that ends a unit of work and updates the database such that other processes can access any changes made.

Common Gateway Interface. A standard protocol through which a Web server can execute programs running on the server machine. CGI programs are executed in response to requests from Web client browsers.

Common Object Request Broker Architecture (CORBA). A middleware specification which defines a software bus—the Object Request Broker (ORB)—that provides the infrastructure

Common User Access (CUA). An IBM architecture for designing graphical user interfaces by use of a set of standard components and terminology.

communications area (COMMAREA). In a CICS transaction program, a group of records that describes both the format and volume of data used.

component model. An architecture and an API that allows developers to define reusable segments of code that can be combined to create a program. VisualAge for Java uses the JavaBeans component model.

composite bean. A bean that is composed of a bean and one or more subbeans. A composite bean can contain visual beans, nonvisual beans, or both.

See also *nonvisual bean*, *bean*, *subbean*, and *visual bean*.

compound document. A means for integrating arbitrary or unstructured data from different sources into one centralized location.

Compound Object Model (COM). The underlying model for all OLE services. It consists of a variety of APIs and object interfaces that allow container components to communicate and interact with one another.

concrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. In the Visual Composition Editor, a visual link between two components that represents the relationship between the components. Each connection has a source, a target, and other properties.

See also *property-to-property connection*, *property-to-method connection*, *script connection*, *event-to-method connection*, *event-to-property connection*, and *parameter connection*.

Console. In VisualAge for Java, the window that acts as the standard input (System.in) and standard output (System.out) device for programs running in the VisualAge for Java IDE.

const. An attribute of a data object that declares that the object cannot be changed.

construction from parts. A software development technology in which applications are assembled from existing and reusable software components, known as parts. In VisualAge for Java, parts are called beans.

constructor. A special class method that has the same name as the class and is used to construct and possibly initialize objects of its class type.

container. A component that can hold other component. In Java, examples of containers include applets, frames, and dialogs. In the Visual Composition Editor, containers can be graphically represented and generated.

CUA. See *Common User Access*.

current edition. The edition of a program element that is currently in the workspace. See also *open edition*.

cursor. A database control structure used by the Data Access Builder to point to specific row within some ordered set of rows and to retrieve rows from a set, possibly making updates or deletions.

cursored emphasis. The appellation of a choice when the selection cursor is on that choice.

D

data abstraction. A data type with a private representation and a public set of operations. The Java language uses the concept of classes to implement data abstraction.

Data Access Builder. A VisualAge for Java Enterprise tool that generates beans to access and manipulate the content of JDBC/ODBC-compliant realtion databases.

database. (1) A systematized collection of data that can be accessed and operated upon by an information processing system. (2) A collection of information such as tables, views, and indexes.

database management system (DBMS). A computer program that manages data by providing the services of centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency control, privacy, and security.

data member. Private data that belongs to a given object and is hidden from direct access by all other objects. Data members can only be accessed by the methods of the defining class and its subclasses.

data object. A storage area used to hold a value.

DB2 Call Level Interface (CLI). The DB2 call level interface is an alternative SQL interface for the DB2 family of products and takes full advantage of DB2 capability. This implementation closely follows industry standards, such as X/OPEN™, to enhance application portability. Currently, the DB2 Call Level Interface functions are compatible with ODBC 2.0, and contain DB2 specific APIs to help exploit DB2 capability.

DB2 for MVS/ESA. An IBM relational database management system for the MVS operating system.

DBCS. See double-byte character set.

DBMS. See *database management system*.

declaration. A description that makes an external object or function available to a function or a block.

default package. In the VisualAge for Java IDE, the package in a project that has no user-defined name. There can only be one default package in a project. For example, imported class files that do not specify a package are placed in the default package.

derivation. The creation of a new or abstract class from an existing or base class.

derived class. A class that inherits from a base class. You can add new data members and members functions to the derived class. You can manipulate a derived class object as if it were a base class object. The derived class can override virtual functions of the base class.

Synonym for *child class* and *subclass*.

destructor. A special class method that has the same name as the class and is used to destroy class objects.

DLL. See *dynamic link library*.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols that can be represented by 256 code points, require double-byte character sets.

Compare to *SBCS*.

dynamic link library (DLL). A file containing executable code and data bound to a program at run time rather than at link time. The C++ Access Builder generates beans and C++ wrappers that let your Java programs access C++ DLLs.

E

edition. A specific “cut” of a program element. VisualAge for Java supports multiple editions of program elements.

See also *open edition*, *versioned edition*, and *current edition*.

encapsulation. The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

Enterprise Access Builders (EAB). In VisualAge for Java Enterprise, a set of code-generation tools.

See also *C++ Access Builder*, *CICS Access Builder*, *Data Access Builder*, and *RMI Access Builder*.

event. An action by a user program, or a specification of a notification that may trigger specific behavior. In JDK 1.1, events notify the relevant listener classes to take appropriate actions.

Compare to *method*, *field*, and *bean event*.

event-to-method connection. A connection from an event generated by a bean to a method of another bean. When the connected event occurs, the method is executed.

See also *connection*.

event-to-property connection. A connection that changes the value of a property when a certain event occurs.

See also *connection*.

F

feature. (1) A major component of a software product that can be installed separately. (2) In VisualAge for Java, a method, field, or event that is available from a bean's interface and that other beans can connect to.

field. A data object in a class. For example, a customer class could have a name field and an address field. A field can itself be an object with its own behavior and fields. By default, a field, contrary to property, does not support event notification.

Compare to *method* and *event*.

free-form surface. The large open area of the Visual Composition Editor where you can work with visual and nonvisual beans. You add, remove, and connect beans on the free-form surface.

framework. A set of cooperative classes with strong connections that provide a template for development.

See also *notification framework*. Compare to *class library*.

FTP. (File Transfer Protocol) The basic Internet function that enables files to be transferred between computers. You can use it to download files from a remote, host computer, as well as to upload files from your computer to a remote, host computer. See Anonymous FTP.

G

garbage collection. A Smalltalk process for periodically identifying unreferenced objects and deallocating their memory.

gateway. A host computer that connects networks that communicate in different languages. For example, a gateway connects a company's LAN to the Internet.

GIF. (Graphics Interchange Format) A graphics file format that is commonly used on the Internet to provide graphics images in Web pages.

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

GUI. See *graphical user interface*.

H

handle. (1) In the Visual Composition Editor, Small square that appear on the corner of a selected visual bean in the free-form surface. Handles are used to resize beans. (2) In Java, a reference to an object in the memory of the Java virtual machine.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

HTML (hypertext markup language). The basic language that is used to build hypertext documents on the World Wide Web. It is used in basic, plain ASCII-text documents, but when those documents are interpreted (called rendering) by a Web browser such as Netscape, the document can display formatted text, color, a variety of fonts, graphic images, special effects, hypertext jumps to other Internet locations and information forms.

HTTP (hypertext transfer protocol). The protocol for moving hypertext files across the Internet. Requires a HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web (WWW). See also Client, Server, WWW.

HTTP request. A transaction initiated by a Web browser and adhering to HTTP. The server usually responds with HTML data, but can send other kinds of objects as well.

hypertext. Text in a document that contains a hidden link to other text. You can click a mouse on a hypertext word and it will take you to the text designated in the link. Hypertext is used in Windows help programs and CD encyclopedias to jump to related references elsewhere within the same document. The wonderful thing about hypertext, however, is its ability to link— using HTTP over the Web — to any Web document in the world, yet still require only a single mouse click to jump clear around the world.

I

inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and methods defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows one to use existing classes as bases for creating other classes.

inner class. In JDK 1.1, class defined in another class.

instance. Synonym for *object*, a particular instantiation of a data type.

Inspector. In VisualAge for Java, a window in which you can inspect and evaluate code fragments in the context of an object.

Integrated Development Environment (IDE). In VisualAge for Java, the set of windows which provide the user with access to development tools. The primary windows are Workbench, Log, Console, Debugger, and Repository Explorer.

Interchange File. A file that you can export from VisualAge for Java, which contains information on selected projects or packages. This file can then be imported into any VisualAge for Java session.

interface. A set of methods that can be accessed by any class in the class hierarchy. The Interface page in the Workbench lists all interfaces in the workspace.

Internet. The vast collection of interconnected networks that all use the TCP/IP protocols and that evolved from the ARPANET of the late 1960's and early 1970's. By July of 1995, the Internet was connecting roughly 60,000 independent networks into a vast global net.

intranet. A private *network* inside a company or organization that uses the same kinds of software that you would find on the public *Internet*, but that is only for internal use. As the Internet has become more popular, many of the tools used on the Internet

are being used in private networks, for example, many companies have Web servers that are available only to employees.

IP. (Internet Protocol) The rules that provide basic Internet functions. See TCP/IP.

IP Number. An Internet address that is a unique number consisting of four parts separated by dots, sometimes called a *dotted quad*. (For example: 198.204.112.1) Every Internet computer has an IP number and most computers also have one or more domain names that are plain language substitutes for the dotted quad.

ISDN. (Integrated Services Digital Network) A set of communications standards that enable a single phone line or optical cable to carry voice, digital network services and video. ISDN is intended to eventually replace our standard telephone system.

iterative development. A software development process that allows progress in stages. At the end of each stage, the result is verified by end users. Through such verification, requirements are dynamically identified and refined while the product is under development.

J

Java. Java is a new programming language invented by Sun Microsystems that is specifically designed for writing programs that can be safely downloaded to your computer through the Internet and immediately run without fear of viruses or other harm to your computer or files. Using small Java programs (called *applets*, Web pages can include functions such as animations, calculators, and other fancy tricks. We can expect to see a huge variety of features added to the Web using Java, since you can write a Java program to do almost anything a regular computer program can do, and then include that Java program in a Web page.

Java archive (JAR). A platform-independent file format that groups many files into one. JAR files are used for compression, reduced download time, and security. Because the JAR format is written in Java, JAR files are fully extensible.

JavaBeans. In JDK 1.1, the specification that defines the platform-neutral component model used to represent parts. Instances of JavaBeans (often called beans) may have methods, properties, and events.

JavaObjs. In VisualAge for Java Enterprise edition, The name of the user-defined default file that contains a list of server objects to be instantiated when the Remote Object Instance Manager is started.

Java Database Connectivity (JDBC). In JDK 1.1, the specification that defines an API that enables programs to access databases that comply with this standard.

Java Native Interface (JNI). In JDK 1.1, the specification that defines a standard naming and calling convention so that the Java Virtual Machine (VM) can locate and invoke methods written in a language different than Java.

See also *native method*.

JAR. See *Java archive*.

JDBC. See *Java Database Connectivity*.

JNI. See *Java Native Interface*.

K

keyword. A predefined word reserved for the Java, that may not be used as an identifier.

L

LAN. Local area network. A computer network located on a user's establishment within a limited geographical area. A LAN

typically consists of one or more server machines providing services to a number of client workstations. See also Ethernet.

legacy code. Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces; usually they are written in a non-object-oriented language, such as C or COBOL.

listener. In JDK 1.1, a class that receives and handles events.

literal. An object that can be created by the compiler. A literal can be a number, a character string, a single character, a symbol, or an array. All literals are unique: Two literals with the same value refer to the same object. The object created by a literal is read-only; it cannot be changed.

literal text. Text in an HTML Text part that is passed to the client browser exactly as entered. You can use literal text to code HTML tagging that is not directly supported by the Web Connection parts.

loaded. The state of the mouse pointer between the time one selects a bean from the beans palette and deposits the bean on the free-form surface.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions.

Log. In VisualAge for Java, the window that displays messages and warnings during development.

M

main bean. The bean that users see when they start an application. This is the bean which has a `main()` method. The main bean is a special kind of composite bean.

See also *bean* and *subbean*.

mapping. See *schema mapping*.

member. (1) A data object in a structure or a union. (2) In Java, classes and structures can also contain functions and types as members.

method. A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

member function. An operator or method that is declared as a member of a class.

method call. A communication from one object to another that requests the receiving object to execute a method.

A method call consists of a method name that indicates the requested method and the arguments to be used in executing the method. The method call always returns some object to the requesting object as the result of performing the method.

Synonym for *message*.

method name. The component of a method call that specifies the requested operation.

message. A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any arguments the method needs for implementation.

Synonym for *method call*.

model. A nonvisual bean that represents the state and behavior of an object, such as a customer or an account.

Contrast with *view*.

N

native method. Method written in a language different than Java that can be called by a Java object using the JNI specification.

named package. In the VisualAge for Java IDE, a package that has been explicitly named and created.

Contrast with *default package*.

nested class. A class defined within the scope of another class.

nonvisual bean. In the Visual Composition Editor, a bean that has no visual representation at run time. A nonvisual bean typically represents some real-world object that exists in the business environment.

Compare to *model*. Contrast with *view* and *visual bean*.

notification framework. In JDK 1.1, a set of classes that implement the *notifier/listener* protocol. The notification framework is the base of the construction from beans technology (Visual Composition Editor).

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and methods that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational methods. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the object class, though it has unique values assigned to its attributes.

object class. A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

object factory. A nonvisual bean capable of dynamically creating new instances of a specified bean. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

object-oriented programming (OOP). A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

ODBC. See *Open Database Connectivity*.

ODBC Driver. An ODBC driver is a dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

ODBC Driver Manager. The ODBC driver manager, provided by Microsoft™, is a DLL with an import library. The primary purpose of the Driver Manager is to load ODBC drivers. The Driver Manager also:

- ❑ provides entry points to ODBC functions for each driver
- ❑ provides parameter validation and sequence validation for ODBC calls.

Open Database Connectivity (ODBC). A Microsoft™ developed C database application programming interface (API) that allows access to database management systems calling callable SQL, which does not require the use of a SQL preprocessor. In addition, ODBC provides an architecture which allows users to add modules called database drivers that link the application to their choice of database management systems at run time. This means applications no longer need to be directly linked to the modules of all the database management systems that are supported.

open edition. An edition of a program element that can still be modified; that is, the edition has not been versioned. An open edition may reside in the workspace as well as in the repository.

operation. A method or service that can be requested of an object.

Object Request Broker (ORB). A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

overloading. An object-oriented programming technique that allows redefinition of methods when the methods are used with class types.

P

package. A program element that contains related classes and interfaces.

See also *default package* and *named package*.

palette. See *beans palette*.

parameter connection. A connection that satisfies a parameter of an action or method by supplying either a property's value or the return value of an action, method, or script. The parameter is always the source of the connection.

See also *connection*.

parent class. The class from which another bean or class inherits data, methods, or both.

part. An existing, reusable software component. In VisualAge for Java, all parts created with the Visual Composition Editor conform to the JavaBeans component model, and are referred to as beans.

See also *visual bean* and *nonvisual bean*. Compare to *Class Editor* and *Composition Editor*.

primary selection. In the Visual Composition Editor, the bean used as a base for an action that affects several beans. For example, an alignment tool will align all selected beans with the primary selection. Primary

selection is indicated by closed (solid) selection handles, whereas the other selected beans have open selection handles.

See also *selection handles*.

primitive bean. A basic building block of other beans. A primitive bean can be relatively complex in terms of the function it provides.

private. In Java, this access modifier is associated with a class member and allows only the class itself to access the member.

process. A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

program. In VisualAge for Java, a term that refers to both Java applets and applications.

program element. In VisualAge for Java, a generic term for a project, package, class, interface, or method.

project. In VisualAge for Java, the topmost kind of program element. A project contains Java packages.

promote features. Make features of a subbean available to be used for making connections. This applies to subbeans that are to be included in other beans, for example, a subbean consisting of three push buttons on a panel. If this example subbean is placed in a frame, the features of the push buttons would have to be promoted to make them available from within the frame.

property. An initial setting or characteristic of a bean. For example, a name, font, text, or positional characteristic.

property sheet. In the Visual Composition Editor, a set of name-value pairs that specify the initial appearance and other bean characteristics. A bean's property sheet can be viewed from the Properties secondary window.

property-to-method connection. A connection that calls a method whenever a property's value changes. It is similar to an event-to-method connection because the property's event ID is used to notify the method when the value of the property changes.

See also *connection*. Compare to *event-to-property connection*.

property-to-property connection. A connection from an property of one bean to a property of another bean. When one property is updated, the other property is updated automatically.

See also *connection*.

property-to-method connection. A connection from a property of a bean to a method. When the property undergoes a state change, then the method is called.

See also *connection*.

protected. In Java, this access modifier is associated with a class member and allows the class itself, subclasses, and all classes in the same package to access the member.

protocol. (1) The set of all messages to which an object will respond. (2) Specification of the structure and meaning (the semantics) of messages that are exchanged between a client and a server. (3) Computer rules that provide uniform specifications so that computer hardware and operating systems can communicate. It's similar to the way that mail, in countries around the world, is addressed in the same basic format so that postal workers know where to find the recipient's address, the sender's return address and the postage stamp. Regardless of the underlying language, the basic protocols remain the same.

prototype. A method declaration or definition that includes both the return type of the method and the types of its arguments.

proxy. An application gateway from one network to another for a specific network application like Telnet or FTP, for example, a firewall's proxy Telnet server performs

authentication of the user and then lets the traffic flow through the proxy as if it were not there. Function is performed in the firewall and not in the client workstation, causing more load in the firewall. Compare with socks.

proxy-bean. A group of client-side and server-side objects that represent a remote server bean. The top-level class that implements the proxy bean is the client-side server proxy.

See *client-side server proxy* and *server-side server proxy*.

R

receiver. The object that receives a method call.

Contrast with *caller*.

Remote Object Instance Manager. Creates and manages instances of server beans through their associated server-side server proxies.

repository. In VisualAge for Java, the storage area, separate from the workspace, which contains all editions (both open and versioned) of all program elements that have ever been in the workspace, including the current editions that are in the workspace. You can add editions of program elements to the workspace from the repository.

Repository Explorer. In VisualAge for Java, the window from which you can view and compare editions of program elements that are in the repository.

resource file. A non-code file that may be referred to from your Java program in VisualAge for Java. Examples include graphic and audio files.

RMI (Remote Method Invocation). In JDK 1.1, the API that allows you to write distributed Java programs, allowing methods of remote Java objects to be accessed from other Java virtual machines.

RMI Access Builder. A VisualAge for Java Enterprise tool that generates proxy beans and associated classes and interfaces so you can distribute code for remote access, enabling Java-to-Java solutions.

RMI compiler. The compiler that generates stub and skeleton files that facilitate RMI communication. This compiler can be automatically invoked by the RMI Access Builder, and can also be invoked off of the Tools menu item.

RMI registry. A server program that allows remote clients to get a reference to a server bean.

roll back. The process of restoring data changed by SQL statements to the state at its last commit point.

runtime type information (RTTI). In JDK 1.1, a set of classes that allows to retrieve the type of an object at run time, and to perform safe casting operations.

S

SBCS. See *single-byte character set*

schema. In the Data Access Builder, the representation of the database that will be mapped.

schema mapping. In the Data Access Builder, a set of definitions for all attributes matching all the columns for your database table, view, or SQL statement. The mapping contains the information required by the Data Access Builder to generate Java classes.

Scrapbook. In VisualAge for Java, the window from which you can write and test fragments of code, without having to define an encompassing class or method.

selection handles. In the Visual Composition Editor, small squares that appear on the corners of a selected visual bean. Selection handles are used to resize beans.

See also *primary selection*.

server. A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

server bean. The bean that is distributed using RMI services and is deployed on a server.

server-side server proxy. Generated by the RMI Access Builder, a companion class to the client-side server proxy, facilitating client-side server proxy communication over RMI.

See also *proxy bean* and *client-side server proxy*.

service. A specific behavior that an object is responsible for exhibiting.

single-byte character set. A set of characters in which each character is represented by a 1-byte code.

SmartGuide. In IBM software products, an interface that guides you through performing common tasks.

SQL predicate. The conditional part of an SQL statement.

stackframe. In the VisualAge for Java Debugger, one of several methods that form a stack trace. Stackframes are listed in the Methods pane, in reverse chronological order; that is, the most recent method called is at the top.

sticky. In the Visual Composition Editor, the mode that enables an application developer to add multiple beans of the same class (for example, three push buttons) without going back and forth between the beans palette and the free-form surface.

stored procedure. A procedure that is part of a relational database. The Data Access Builder can generate Java code that accesses stored procedures.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

subbean. A bean that is used to create another bean.

See also *nonvisual bean*, *bean*, and *visual bean*.

superclass. See *abstract class* and *base class*.

T

TCP/IP. (Transmission Control Protocol/Internet Protocol) The basic programming foundation that carries computer messages around the globe via the Internet. The suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

tear-off property. A property that a developer has exposed to work with as though it were a stand-alone bean.

thread. A unit of execution within a process.

tool bar. The strip of icons along the top of the free-form surface. The toolbar contains tools to help an application developer construct composite beans.

transaction. In a CICS program, an event that queries or modifies a database that resides on a CICS server.

type. In VisualAge for Java, a generic term for a class or interface.

U

UI. See *user interface*.

Unicode. The Unicode is a character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern

world. Unicode characters are normally encoded using 16-bit integral unsigned numbers.

unifor resource locator (URL). A standard identifier for a resource on the World Wide Web, used by Web browser to initiate a connection. The URL includes the communications protocol to use, the name of the server, and path information identifying the objects to be retrieved on the server. A URL looks like this:

`http://www.matisse.net/seminars.html`
or `telnet://well.sf.ca.us.br`
or `news:new.newusers.question.br`

unloaded. The state of the mouse pointer before a user selects a bean from the beans palette and after the user deposits a bean on the free-form surface. In addition, a user can unload the mouse pointer by pressing the Esc key.

unresolved problem. In VisualAge for Java, an error marked by an X to the left of the program element name. For example, a declaration or definition problem. You can view all unresolved problems from the Unresolved Problems page in the Workbench.

user interface (UI). (1) The hardware, software, or both that enables a user to interact with a computer. (2) The term *user interface* normally refers to the visual presentation and its underlying software with which a user interacts.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A bean that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a bean elsewhere in the application.

version. The act of making an open edition a versioned edition; that is, making the edition read-only.

versioned edition. An edition that has been versioned and can no longer be modified.

view. (1) A visual bean, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces.

Compare to *visual bean*. Contrast with *model*.

visual bean. In the Visual Composition Editor, a bean that is visible to the end user in the graphical user interface.

Compare to *view*.

Contrast with *nonvisual bean*.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

Visual Composition Editor. In VisualAge for Java, the tool where you can create graphical user interfaces from prefabricated beans, and define relationships (called connections) between both visual and nonvisual beans. The Visual Composition Editor is a page in the class browser.

W

white space. Space characters, tab characters, form-feed characters, and new-line characters.

window. (1) A rectangular area of the screen with visible boundaries in which information is displayed. Windows can overlap on the screen, giving it the appearance of one window being on top of another. (2) In the Visual

Composition Editor, a bean that can be used as a container for other visual beans, such as push buttons.

Workbench. In VisualAge for Java, the main window from which you can manage the workspace, create and modify code, and open browsers and other tools.

workspace. The work area that contains all the code you are currently working on (that is, current editions). The workspace also contains the standard Java class libraries and other class libraries.

List of Abbreviations

ANSI	American National Standards Institute	IBM	International Business Machines Corporation
APA	all points addressable	IDE	integrated development environment
API	application programming interface	ISO	International Organization for Standardization
ATM	automated teller machine	ITSO	International Technical Support Organization
CAD	computer-aided design	IWO	IBM Workframe option project
CAE	Client Access Enabler	JAR	Java archive
CGI	Common Gateway Interface	JDBC	Java Database Connectivity
CICS	customer information control	JDK	Java developer's kit
CLI	call level interface	JNI	Java Native Interface
COM	Compound Object Model	LAN	local area network
CORBA	Common Object Request Broker Architecture	MVS	multiple virtual storage
CRC	class-responsibility-collaborator	NLS	National Language Support
CUA	Common User Access	NT	new technology
DB2	DATABASE 2	NTSF	Windows NT™ file system
DBCS	double-byte character set	ODBC	Open Database Connectivity
DBMS	database management system	OMF	object module format
DLL	dynamic link library	OMG	Object Management Group
DNS	domain name system	OMT	object modeling technique
DRDA	Distributed Relational Database Architecture	OO	object-oriented
FAT	File Allocation Table	OOA	object-oriented analysis
FTP	File Transport Protocol	OOD	object-oriented design
GUI	graphical user interface	OOP	object-oriented programming
HPFS	high-performance file system	OOSE	object-oriented software engineering
HTML	Hypertext Markup Language	ORB	object request broker
HTTP	Hypertext Transfer Protocol	OS/2	Operating System/2
HTTDP	Hypertext Transfer Protocol daemon	PM	Presentation Manager
		RAD	rapid application development

<i>RDBMS</i>	relational database management system
<i>RDD</i>	responsibility-driven design
<i>RMI</i>	remote method invocation
<i>RTTI</i>	run-time type identification
<i>SBCS</i>	single-byte character set
<i>SDK</i>	software developer kit
<i>SQL</i>	Structured Query Language
<i>SSL</i>	secure sockets layer
<i>SVGA</i>	super video graphics array/adaptor
<i>TCP/IP</i>	Transmission Control Protocol/Internet Protocol
<i>UI</i>	user interface
<i>URL</i>	uniform resource locator
<i>VMT</i>	visual modeling technique
<i>WWW</i>	World Wide Web

Index

A

- abstract 128
- abstract class 17, 107
- abstract method 17, 108
- Abstract Windowing Toolkit. See AWT
- abstraction 43
- access modifier
 - default 17
 - package 22
 - private 17
 - protected 17
 - public 17
- accountId 63
- ActionListener interface 210, 224
- action() method 206
- ActiveX 230
- actor 48
- adapter 213
- aggregation 115–118, 133, 172
- ALIGN 397
- ALT 397
- applet
 - concept 5, 201
 - converting 344
- APPLET tag 396
- Applet Viewer 8, 220
- application
 - analysis 48–50
 - building your first 13
 - concept 202
 - design 50
 - problem domain 48
- ARCHIVE 397, 398
- array 31
- ArrayIndexOutOfBoundsException class 31
- association 118
- asynchInvokeTxn() method 420, 421
- ATM 41, 47, 48, 292
- AtmAccountPanel class 300–309
- AtmCard class 278
- AtmHistoryPanel class 327–329
- AtmKeypad class 309–312
- AtmKeypadPanel class 313–315
- AtmLanguagePanel class 376–379
- AtmPanel class 321–326
- AtmPinPanel class 319–320
- AtmWelcomePanel class 315–318, 379
- AUTOEXEC.BAT 402

- automated teller machine. See ATM
- AWT 198–232

B

- backoutUOW() method 421
- balance 62
- BankAccount
 - definition 47, 51
 - serialization 174
 - versioning 192
- bean
 - adding to the free-form surface 236
 - assembling 329
 - customizing 237
 - event 231, 247
 - feature 246
 - method 249
 - persistence 250
 - promoting a feature 308, 315, 317
 - property 248
 - reusing 296, 314
 - variable 304
- BeanInfo class 250, 286, 434
- BeanInfo page 234, 251, 252
- beans palette 234
- bibliography 451
- binary file 394
- black box 230
- block 21
- BorderLayout class 216, 298, 313, 321
- bound property 248
- break 35
- breakpoint
 - inserting 157
 - removing 157
- Browse pane 10
- BufferedInputStream class 166
- BufferedOutputStream class 167
- BufferedReader class 166
- BufferedWriter class 167
- Button class 203, 239
- ByteArrayInputStream class 165
- ByteArrayOutputStream class 166
- bytecode 19

C

- calculator 251–274
- call stack 144
- CardLayout class 217, 298, 331

- case 33
- casting 24
- catch block 141, 148
- CD-ROM 5, 401, 413
- CGI 405
- character
 - property 373
 - type 374
- CharArrayReader class 165
- CharArrayWriter class 166
- CheckboxMenuItem class 224
- CheckingAccount class 106
- ChoiceFormat class 368
- CICS 419, 439
- class
 - abstract 107, 115, 128
 - blueprint 46
 - candidate 49
 - concept 5, 46, 104
 - concrete 108
 - creating 51, 60
 - icon 16
 - implementing 60
 - inner 118
 - method 43
 - modifier 52, 72
 - testing 69
 - variable 43
 - versioning 359
- class browser 233
- Class class 387
- Classes view 10
- CLASSPATH 73, 398, 427
- client-side server proxy 431
- clone 94
- clone() method 106
- COBOL 420
- CODE 397
- code
 - beanizing 275, 281, 283
 - editing 4
 - exporting 81, 394
 - importing 78, 80, 274
- CODEBASE 397
- collation 371
- CollationKey class 372
- Collator class 371
- collision 360
- COMMAREA 420, 439
- comment 29
- commitUOW() method 421
- Common Gateway Interface. See CGI
- compatibility 410
- component 198, 230
- Component class 199

- conditional statement
 - if-else 32
 - switch 33
- connection
 - event-to-method 261–263
 - event-to-property 269
 - event-to-script 311
 - parameter 266–269
 - property-to-method 269
 - property-to-property 257–261
 - reordering 326
 - script 263–266
- Console window 15
- constrained property 249
- construction from parts 230
- constructor
 - concept 87
 - copy 90
 - modifier 94
 - overloading 89
 - overriding 88
- container 200
- Container class 214
- continue 35
- CPPFILT 430
- credit() method 66
- currency formatting 369
- customer information control. See CICS
- C++ 21
- C++ Access Builder 424

D

- DAException class 418
- DAIOStream class 418
- DAManager class 418
- Data Access Builder 415, 439
- database management system. See DBMS
- DataInputStream class 166
- DataOutputStream class 167
- DatastoreJDBC class 417
- date formatting 370
- DB2 4, 414
- DBMS 438
- debit() method 66
- Debugger 4, 153, 156
- DecimalFormat class 368
- defineClass() method 387
- delegation 117, 141
- delegation model 205
- deprecated method 207
- destroy() method 202
- destructor 95

- Dialog class 203
- dispose() method 206, 239
- DLL 430
- do loop 34
- domain name 401
- dynamic binding 111, 387

E

- EAB 4, 414
- early binding 111
- ECI 4, 414
- editable property 249
- edition
 - concept 182, 184
 - creating 187
 - loading 188
 - managing 187
 - replacing 187
- editor 4
- encapsulation 44, 53, 230
- endUOW() method 421
- Enterprise Access Builder. See EAB
- Enterprise Edition 413
- Entry Edition 413
- equals 106
- error
 - handling 140
 - icon 18
 - semantic 65
 - syntactic 65
 - unresolved 66
- Error class 146
- event
 - creating 338
 - handling 205
 - model 231
 - source 209, 306, 342
- event listener 209
- EventListener interface 339
- event-to-method connection 261–263
- event-to-property connection 269
- event-to-script connection 311
- exception
 - catching 142, 148
 - concept 141
 - defining 149
 - displaying 336
 - hierarchy 146
 - object 144
 - predefined 148
 - propagating 145

- throwing 142
- Exception class 146, 148
- exception handler 25, 142
- executable 17
- execution flow 143
- execution stack 144
- expression 21
- extends 106
- external call interface. See ECI 414
- Externalizable interface 250

F

- Factory bean 346
- field
 - concept 52
 - initializing 86, 87
 - modifier 52
 - transient 17
- File class 167
- FileInputStream class 165, 176
- FileOutputStream class 166
- FileReader class 165
- FileWriter class 166
- final 52, 88
- finalizer 94, 143
- finalize() method 95, 100, 106
- finally block 143
- FlowLayout class 215, 298, 322
- Frame class 203, 239
- free-form surface 235

G

- garbage collector 95
- generalization 104
- generated code 241
- getDateInstance() method 370
- getDateTimeInstance() method 370
- getter method 53, 63
- GridBagConstraints class 221
- GridBagLayout class 221, 299, 301, 316, 319, 321, 328, 377
- GridLayout class 218, 299, 310, 322
- GUI 198

H

- handle 46

handleEvent() method 207
heap 92, 94
HEIGHT 397
HiAgain class 13
HiThere class 7
HotJava 408
HSPACE 397
HTML 396, 399, 403
HTML page
 creating 396
HTTP
 configuring 404
 port number 406
 starting server 403
httpd.cnf 407
Hypertext Markup Language. See HTML

I

IBM Internet Connection Server 400
 administrator 404
 home directory 405
 installing 401
 requirements 401
 starting server 403
IBM Open Class Library 361
IBM VisualAge for C++ 425
IDE 4, 6, 66, 183
if-else 32
IList class 315
ImageCanvas class 375
IMessageBox class 336
implementation phase 50
import 74, 78
incremental development 106
indexed property 248
inheritance 47, 104–114, 121
init() method 201
inner class 118, 209
InputStream class 165, 171
Inspector 153, 154
instance method 43
instance variable 43
Integer class 56
integrated development environment. See IDE
interchange file 79, 394
interface 17, 112, 114, 130
Interfaces view 10
international application 375
Internet xii
Internet Connection Server. See IBM Internet Connection Server

Internet Explorer. See Microsoft Internet Explorer
introspection 231, 246, 250
Introspector class 250
invokeTxn() method 420, 421
iterative statement
 do-while 34
 while 33
IVector class 284
ivj2cpp 427
IVJCicsUOWInterface class 421, 422

J

J2CPP_CLASSPATH 427
JAR 79, 250, 395
Java archive. See JAR
Java Database Connection. See JDBC
Java native interface. See JNI
Java virtual machine. See Java VM
Java VM 3, 19
JavaBeans 4, 198, 228, 229, 247, 415
JDBC 4, 414
JDBC-ODBC Bridge 414
JNI 424

K

keyword
 abstract 109
 break 35
 case 33
 catch 141
 continue 35
 do 34
 extends 106
 final 117
 finally 143
 if-else 32
 import 74, 78
 interface 112
 new 85
 null 26
 package 73
 protected 109
 super 107, 121
 switch 33
 synchronized 360
 this 90
 throw 145
 transient 172

- try 142
- while 33

L

- layout constraint 320
- layout manager 204, 214, 298
- library 83, 417
- LineNumberInputStream class 166
- LineNumberReader class 166
- Listener interface 340
- ListResourceBundle class 365, 367
- literal 24
- locale 362
- Locale class 362

M

- maintainability 45
- main() method 13, 14, 202
- menu 224, 270
- menu bar 270
- Menu class 224, 270
- MenuBar class 224
- MenuComponent class 200, 224
- MenuItem class 224, 270
- MenuSeparator class 271
- message 5, 45, 59
- message formatting 368
- MessageFormat class 368
- method
 - abstract 108
 - argument 93
 - candidate 49
 - concept 5
 - final 17
 - getter 53, 63
 - modifier 54
 - name 55
 - native 17
 - overloading 121
 - overriding 107
 - parameter 55
 - return type 55
 - setter 53, 63
 - signature 55, 88, 89
 - static 17
 - synchronized 17
- method modifier
 - private 54
 - protected 54

- public 54
- Microsoft Internet Explorer 410
- Microsoft Visual C++ 425
- MIME-type 407
- model 231, 252
- model/view/controller. See MVC
- modifier
 - final 22, 88, 117
 - private 22, 72
 - protected 22, 72
 - public 22, 72
 - static 22
 - transient 22
 - volatile 22
- modularity 44
- MouseEvent class 227
- MS-DOS 401
- multilingual application 375
- multiple inheritance 114
- multiple virtual storage. See MVS
- multiplicity 116
- multithreading 21, 356
- MVC 208
- MVS 420

N

- NAME 397
- naming convention 62
- National Language Support. See NLS
- Netscape Communicator 409
- NLS 361–386
- nonvisual bean 232
- null 26
- Null layout 299
- number formatting 368
- NumberFormat class 364, 368, 369

O

- object
 - assignment 59
 - behavior 42
 - cloning 94
 - concept 5, 42
 - design 50
 - downcasting 389
 - flattening 164
 - handle 46
 - interface 52
 - method 42

- reference 58, 90, 91, 93
- resurrecting 164
- serialization 163, 170–173
- using 58
- variable 42
- Object class 214
- ObjectInputStream class 171, 173, 176
- ObjectOutputStream class 171, 173
- OpenDoc 230
- operator
 - binary 26
 - member access 28
 - precedence 28
 - relational 27
 - unary 26
- OutputStream class 171
- OverdraftAccount class 109

P

- package 52
 - access 22
 - concept 5, 71
 - copying 96
 - creating 73, 76
 - icon 16
 - using 74, 77
- Packages view 10
- paint() method 202
- Panel class 201
- PARAM NAME tag 397
- parameter connection 266–269
- percentage formatting 370
- performance 20
- persistence 164, 246
- PinCheckedOkEvent class 338
- PipedInputStream class 165, 166
- PipedOutputStream class 165, 166
- PipedReader class 165, 166
- PipedWriter class 165, 166
- PODataId class 417
- polymorphism 47, 110, 112
- pop-up menu 226, 272
- PopupMenu class 224, 272
- port 406
- portability 20, 246
- PersistentObject class 417
- primary bean 235
- PrintStream class 167
- PrintWriter class 167
- private 52, 54, 72
- problem domain 48
- Professional Edition 4, 413

- program element
 - searching 191
 - versioning 185
- project 5, 16, 82
- Projects view 10
- promote 308, 315, 317
- property
 - concept 231
 - promoting 308
 - tearing off 307
- PropertyResourceBundle class 366
- property-to-method connection 269
- property-to-property connection 257–261
- protected 52, 54, 72
- proxy bean 431
- public 54, 72, 73
- PushbackInputStream class 166
- PushbackReader class 166

Q

- Quick Start dialog 6

R

- RandomAccessFile class 167
- Reader class 165
- reflection 250, 386
- relationship
 - has-a 104
 - is-a 104
- remote method invocation. See RMI
- Remote Object Instance Manager 431, 432
- repository
 - concept 182, 183
 - exploring 189
- resource bundle
 - concept 364
 - loading 366
 - retrieving resource from 367
 - using 380
- resource locking 360
- ResourceBundle class 364, 366
- reuse 103
- RMI 4, 431, 439
- RMI Registry 432
- RTTI 386–390
- Runnable interface 356
- run-time binding 111
- Run-Time Type Identification. See RTTI
- RuntimeException class 147

S

- safety 20
- sand-box 20
- SavingsAccount class 107
- scope 94
- Scrapbook 158
- script connection 263–266
- separator 28
- SequenceInputStream class 165
- Serializable interface 163, 171, 250
- serialization 163
- server-side server proxy 432
- setter method 53, 63
- Smalltalk 21
- SmartGuide 6, 60, 76, 253
- Source pane 10
- specialization 104
- SQL 415, 416
- startUOW() method 421
- start() method 201
- static 57
- static initializer 88
- StaticTicker class 353
- Sticky checkbox 234, 310
- stop() method 201
- stream
 - input 165–166
 - object 170–173
 - output 166–167
 - using 168
- StreamTokenizer class 167
- String 29
- StringBuffer 30
- StringBuffer class 165, 167
- StringBufferInputStream class 165
- StringReader class 165
- StringWriter class 167
- Structured Query Language. See SQL
- subclass 106
- super 107, 121
- switch 33
- symbol 16
- system
 - design 50
 - requirements 48

T

- tabbing order 303
- TCP/IP
 - domain name 401

- host name 401
- IP address 401
- port 406
- tear-off property 306
- TextField class 203, 239
- this 90, 308
- thread 159, 205, 356
- Thread class 356
- throw 145
- Throwable class 146
- Ticker class 358
- time formatting 370
- toolbar 235
- toString() method 90, 100, 106, 289
- Transaction class 117, 174
- transient 171, 172
- try block 142

U

- UML 51, 251
- Unicode 25, 375
- unified modeling language. See UML
- uniform resource locator. See URL
- unit of work 421
- UNIX 175
- unresolved problem 66
- Unresolved Problems view 10
- URL 396
- use case 48

V

- variable
 - access modifier 22
 - declaration 21
 - final 22
 - initial value of 23
 - modifier 22
 - name of 23
 - private 22
 - protected 22
 - public 22
 - scope of 25, 94
 - static 22
 - transient 22
 - type of 22
 - volatile 22
- variable bean 304, 346
- Vector class 56, 116
- version control

- concept 4
 - using 184–190
- view 231, 252, 256, 292
- virtual machine. See Java VM
- visual bean 232
- Visual Composition Editor 197, 232–246, 277
 - concept 4
 - palette 462
- VSPACE 397

W

- Web browser
 - HotJava 408
 - Internet Explorer 410
 - Netscape Communicator 409
 - Netscape Navigator 5
- Web page
 - creating 396
- Web server
 - configuration file 405
 - configuring 404, 404–408
 - home directory 405
 - host name 406
 - installing 401–404
 - port number 406
 - starting 403
- while loop 33
- WHTTPG 403
- WIDTH 397
- WindowListener interface 212
- Windows 95 441
- Windows NT 175, 441
- word break 374
- Workbench 6, 7–10
- workspace 75, 181, 182, 183
- wrapper class 56
- WWW xii